**University of Applied Sciences – Stuttgart**

**Hochschule für Technik - Stuttgart**

# *Design And Implementation Of Temporal Triggers For MySQL RDBMS*

by

**Andrey Jordanov Hristov**

A thesis presented to the
University of Applied Sciences Stuttgart
In fulfilment of the thesis requirement
for the degree of

**Master Of Science**
in
**SOFTWARE TECHNOLOGY**

**Supervisors:**
**Prof. Dorothee Koch – HFT Stuttgart**
**Dr. Sergii Golubchyk – MySQL AB**

**February 12, 2005**

## Acknowledgements

This thesis would not have been possible without the ideas and insights of the following people at MySQL AB and the Stuttgart University of Applied Sciences; to whom I would like to express my appreciations for their efforts and support:

- Dr. Sergii Golubchyk for taking the responsibility of being technical supervisor of this thesis.
- My university supervisor Prof. Dorothee Koch for being my university supervisor.
- Peter Gulutzan of MySQL AB, and author of [SQL99Comp], for helping me read the SQL-99 standard.
- Dmitri Lenev of MySQL AB for the numerous cases he helped me with hints about MySQL internals, whenever my technical supervisor was not available.
- Mike Hillyer, technical writer of MySQL AB, for throughout  review of my final draft.
- Brian Aker and Georg Richter of MySQL AB for offering this master thesis.
- Georg Richter, Kaj Arnö and Patrik Backman for offering me employment at MySQL AB.

And like every coin has two sides, I would like to thank the following people, because without their support this thesis wouldn't be finished:

- Gaylord Aulke and Oliver Schmidt of Dorten GmbH.
- Kirsten Fischer and Georg Richter for their general support.
- My mother and my grandmother for their general support.
- Branimir Hristov giving me as a present my first computer and not only that.
- Yanko Baev and Tsveta Baeva for showing me that mathematics is something beatiful and thus helping me develop my rational thinking.
- Prof. Jordan Hristov for helping me whenever needed.

# Table of Contents

1. Introduction....................................................................5
   1.1 Abstract....................................................................5
   1.2 Objective....................................................................6
   1.3 Short conceptual formulation....................................................6
   1.4 Prototype applications..........................................................7
2.  Temporal triggers in commercial systems..........................................8
   2.1 Oracle 10g....................................................................8
   2.2 MS SQL Server..................................................................12
   2.3 Sybase Adaptive Server Enterprise.............................................16
   2.4 Sybase Adaptive Server IQ.....................................................18
   2.5 IBM DB2 Universal Database...................................................19
   2.6 Summary......................................................................21
3. Design of High Level Architecture...............................................22
   3.1 Detailed requirements.........................................................22
      Types......................................................................22
      Type conversion...............................................................23
      Execution discipline.........................................................23
      Termination...................................................................24
      Ownership.....................................................................24
      Commands executed by temporal triggers.......................................25
      Output logging................................................................25
      Time zones....................................................................26
      Execution plan time restrictions.............................................27
      Privileges....................................................................28
      Metadata storage..............................................................29
      Validation....................................................................30
      Module administration.........................................................30
      Replication of TT..............................................................31
      Run-time statistics:..........................................................32
      Backup and export.............................................................32
      Multithreading................................................................33

Self referencing....................................................................33
Testing.................................................................................33
3.2 Temporal triggers SQL grammar.............................................33
3.3 Summary..............................................................................35
4. Internals of MySQL RDBMS related to the implementation of temporal triggers..................................................................................37
4.1 Client/Server.........................................................................37
4.2 Processes and Multithreading...................................................40
4.3 Memory management............................................................45
4.4 Query processing...................................................................47
4.5 Storage engines.....................................................................50
4.6 System library.......................................................................51
4.7 Access control lists................................................................53
4.8 Stored procedures.................................................................55
4.9 Table triggers........................................................................60
5. Design of Low Level Architecture and Implementation.....................63
5.1 Temporal triggers class...........................................................63
5.2 Parsing of SQL statements related to TTs..................................64
5.3 Metadata storage...................................................................69
5.4 In-memory caching................................................................72
5.5 Multithreading.......................................................................74
5.6 Statistical variables................................................................75
5.7 Server variables.....................................................................76
5.8 Automatic testing...................................................................77
6. Conclusion...............................................................................79
Appendices..................................................................................82
Appendix A. Glossary..................................................................82
Appendix B. References ..............................................................88
Appendix C. MySQL server class diagrams .....................................91

# 1. Introduction

## 1.1 Abstract

There exist on the market dozens of relational database management systems (RBDMS) and every organization that creates such a product tries to add more and more features to provide a superior product. Having more features brings more customers and therefore allows the company to grow and extend the product in different directions.

In Unix and Unix-like operating systems (OS), as well in some versions of the Windows$^{TM}$ OS, there are services that can be used for scheduling the execution of tasks at specific times. In the Unix world the best known of such services are named *crontab* and *at.* In the Windows OS an equivalent exists called the "Task scheduler". These services are usually used to schedule transiently occurring tasks such as backups, system checking and others.

Facilities for scheduling and executing tasks exist also in all major RBDMSes such as IBM DB2, Oracle 9i & 10g, Microsoft SQL Server 7.0 & 2000, Sybase AES and others. The methods for creating and managing the scheduling of the tasks differ but the functionality provided highly correlates.

MySQL is an open source RDBMS developed by the company MySQL AB. This RDBMS has gained much popularity amongst the developers of open source and commercial software, because of the simplicity of use and administration. The so-called learning curve is not as steep as when using commercial alternatives (proven from the wide distribution of MySQL). During the last few years the list of features provided by MySQL has grown significantly.

This master thesis introduces the design and implementation of facilities for scheduling and executing tasks, which later on will be called events or temporal triggers, in the MySQL RDBMS. It must be noted that temporal triggers are not temporary triggers but triggers executed at a certain time, instead of executed on a table event. There are differences between table triggers (also known simply as triggers) and temporal triggers.

## 1.2 Objective

The objective of this master thesis is the creation of a prototype that provides the functionality for scheduling of events and their execution, at specific moments in time, inside the MySQL RDBMS. Being able to automate tasks, by means of their execution without user intervention, is a key component of a well-developed database product. The applications of this feature are many and they include scheduled back-ups, data cleansing and data extraction in the area of Data Warehousing.

## 1.3 Short conceptual formulation

The temporal triggers, and more specifically the Event Executor (EVEX) subsystem, have to provide the functionality for scheduling and execution, by means of events creation, alteration, removal, execution and logging. The creation, alteration and removal of temporal triggers have to use a syntax similar to those specified in SQL-92/SQL-99. The SQL-99 standard (also known as SQL3) lacks specification of a syntax for temporal triggers as entities, and every DBMS manufacturer has to create their own syntax. Triggers are part of SQL3; as are stored procedures (SP), and temporal triggers have many things in common with them. Therefore, the syntax used has to be straightforward and similar to those of the aforementioned.

It should be possible to define a schedule plan that may allow event execution like: "every Monday at 02:00" or "every 15 minutes". Scheduling like "on Dec 23th 2004 at 14:35:00" should also be possible.

Like stored procedures, events must be stored in a system table in the *mysql* catalog.

The implementation must be done in C++ and there are no limitations on the platform on which the development is done. C++ should be the language used for the implementation, because MySQL server itself is implemented in this language and therefore the binding will be as seamless as possible. Moreover, MySQL server already includes a framework that can be used to ease the implementation process.

## 1.4 Prototype applications

The range of applications of the prototype is quite broad but some deserve to be mentioned:

- backup procedures – An user may schedule daily, weekly or other kind of backups, and they will be performed without his/her intervention.
- stale data cleaning – Many web sites keep their session data in relational database systems to ease scalability. However, the nature of Internet/Intranet applications should be taken into consideration; stale session data may reside in the system. A temporal trigger can delete this data at a regular interval.
- data warehousing – Data extraction, as well as other data warehousing procedures can be automated and executed internally, removing reliance on external tools.
- platform independence – Having the temporal triggers functionality built-in makes the MySQL RBDMS independent of external tools that provide the same functionality. Hence, this functionality is offered to users on all platforms supported by the system.
- data checks : system procedures for checking the consistency of the data and the system health could be run automatically at regular intervals.

# 2. Temporal triggers in commercial systems

*"Those who cannot learn from history are doomed to repeat it."*
George Santayana

After the brief introduction in Chapter 1, this chapter continues with a concise review of several commercial relational database systems, which have features similar to those of temporal triggers. The reason for reviewing only commercial systems is not intentional, but to the knowledge of the author there are no non-commercial systems that implement similar features. The following non-commercial RDBMSes were checked:

- PostgreSQL 7.4[1]
- Firebird (Interbase)
- SQLite[2]

In Chapter 3, the high level architecture of the prototype will be based partly on this review. The following commercial products were reviewed:

- Oracle 9i
- Oracle 10g
- MS SQL Server 2000, 2003
- Sybase Adaptive Server Enterprise
- Sybase Adaptive Server IQ
- IBM DB2 Universal Database

This list is not by any means complete, but represents the information known to the author of this thesis.

## 2.1 Oracle 10g

Oracle 10g features a new job-scheduling facility, named "The Scheduler". As in Oracle 9i, the control over the execution of jobs inside the RBDMS is through an interface exposed by a package of stored procedures; namely *dbms_scheduler*. This package deprecates and replaces the package used in Oracle 9i, named *dbms_job*.

"The Scheduler" offers extended functionality over the one provided

[1] The latest stable version by the time of the writing of this thesis
[2] SQLite does not use a client/server approach

in Oracle 9i. A user can execute a variety of code, by means of executing stored procedures in the form of PL/SQL routines (or ones written in another supported language), as well as anonymous SQL blocks, shell scripts and binaries. The binaries are programs outside of the system that are executed by using the OS on which Oracle runs. The shell scripts are small programs written in a scripting language recognized by a command interpreter. They are quite common on Unix platforms and have a shallow equivalent on DOS/Windows in the form of batch files. There are a handful of command interpreters for Unix, and each has its own unique command language, thus making portability a difficult task. Oracle 10g's documentation uses the term *program* to describe all this entities with a single term. A *program* is not just a name, but also a collection of metadata that is needed by the system to identify and execute the entity correctly. For instance, program arguments are part of this metadata.

Oracle 10g uses a two layered model to abstract the creation and administration of jobs from their scheduling and execution. This approach is different than the one-layer approach used by the prototype built for this master thesis.

First, a program is defined as well as its metadata. A schedule, or several schedules, might be created that involve the program but are independent entities. Different users may use a program at different times, eliminating the need of having the program redefined now and then. To perform this, a program has to be stored in a program *library*, which is accessible by other users and permits reuse.

The tuple (program, schedule) is known as *job*. A schedule exists as an entity inside Oracle and is additional metadata attached to a *program*, which makes it a job. In terms of OOP, a *job* extends *program* by adding more metadata needed for the execution of the former.

The user can divide the jobs into *categories* for greater control. "The job class is a category of jobs that share various characteristics, such as resource consumer group assignments and assignments to a common, specific, service name." [ODNF]. Job classes are related to job windows. The latter are related to resource plans, which organize the usage of available resources in a proper manner.

Two different ways of using "The scheduler" are available to suit different needs. The first interface is through the *dbms_scheduler* package and the second one is through a GUI application named Oracle Enterprise Manager (OEM). The latter helps with the creation of jobs by less experienced users or users who prefer using graphical interfaces.

A scenario to create a job consists of one, two or three steps depending on whether the code that should be executed is an anonymous SQL block or an existing program, in the sense explained above. Additionally, the number of steps depends on how the user will specify the schedule plan. Whenever an anonymous SQL block is used, the user may proceed directly to the creation of a job. However, if a program has to be used, then the user must create it as a database entity by using the *create_program()* stored procedure and after that create a job that uses the newly created program. One may create the schedule by directly specifying start_date, end_date and repeat_interval as part of the call to *create_job()* or by specifying an identifier of a saved schedule, which had been created by calling *create_schedule()*. The job creation can be divided in the following steps:

1. Program creation
2. Program arguments definition.
3. Schedule creation (optional)
4. Job creation.
5. Job arguments definition.

To create a program as a database entity one has to use the *dbms_scheduler.create_program()* stored procedure as mentioned above. Note that Oracle is case-insensitive for procedure names. The definition of this procedure is [1, 4]:

```
DBMS_SCHEDULER.CREATE_PROGRAM (
program_name IN VARCHAR2,
program_type IN VARCHAR2,
program_action IN VARCHAR2,
number_of_arguments IN PLS_INTEGER DEFAULT 0,
enabled IN BOOLEAN DEFAULT FALSE,
comments IN VARCHAR2 DEFAULT NULL);
```

All parameters are of type IN.

Description:

- **program_name** : The program's unique identifier.
- **program_type** : The type of executable to be scheduled. Valid values are PLSQL_BLOCK, STORED_PROCEDURE, and EXECUTABLE.
- **program_action** : The name of the executable, stored procedure or an anonymous SQL block.
- **number_of_arguments** : The number of arguments the program expects. Only used for EXECUTABLE programs, since it is not possible to directly identify how many arguments they expect. This parameter is ignored for PLSQL_BLOCK, since the information needed is available to *CREATE_PROCEDURE()* from the stored procedure metadata.
- **enabled** : Specifies whether the program is enabled or not. If a program is disabled (i.e. not enabled) then it is not scheduled for execution. A program that fails a specified number of times during execution is automatically disabled. By default a program is created as *enabled*.
- **comments** : Program description.

Example of a one-step job creation:

```
EXEC dbms_scheduler.create_job(
   job_name   => "delete_stale_sessions",
   job_type   => "PLSQL_BLOCK",
   job_action => "BEGIN DELETE FROM sessions
                        WHERE last_accessed < SYS_DATE – 15;
                        COMMIT;
                  END;",
   start_date     => "15-FEB-2005 01.00.00 AM Europe/Berlin",
   repeat_interval=> 'SYSTIMESTAMP + INTERVAL '15' MINUTE',
   comments       =>"Job which deletes old sessions from DB");
```

This job is executed every hour and executes the anonymous SQL block provided.

[ODAG] states: "*In the case of repeat intervals that are based on PL/SQL expressions, the time zone is part of the timestamp that is returned by the PL/SQL expression. In both cases, it is important to use region names. For example, "Europe/Istanbul", instead of absolute time zone offsets such as "+2:00". Only when a time zone is specified as a region name will the Scheduler follow daylight savings adjustments that apply to that region.*".

Example of a two-step job creation (including schedule creation):

```
EXEC dbms_scheduler.create_schedule(
  schedule_name   => 'sessions_15min_schedule',
  start_date      => SYSTIMESTAMP,
  end_date        => NULL,
  repeat_interval => 'SYSTIMESTAMP + INTERVAL '15' MINUTE',
  comments        => 'Every 15 mins starting now, never expiring.');

EXEC dbms_scheduler.create_job(
   job_name      => 'delete_stale_sessions_with_schedule',
   program_name  => 'delete_stale_session_prog',
   schedule_name => 'sessions_15min_schedule'
);
```

The owner of the job is the user in whose schema the job was created. Conversely, the job creator is the user who created the job.

## 2.2 MS SQL Server

MS SQL Server's interface for working with jobs (the same term as used in Oracle documentation) is again through a set of stored procedures, grouped in a module named “SQL Server Agent Procedures” [MSDN].

The designers of MS SQL have decided to use a three layered model which extends of the one used in Oracle 10g.

A subset of the routines in the module mentioned above [MSDN]:

- sp_add_job

```
sp_add_job [ @job_name = ] 'job_name'
  [ , [ @enabled = ] enabled ]
  [ , [ @description = ] 'description' ]
  [ , [ @start_step_id = ] step_id ]
  [ , [ @category_name = ] 'category' ]
  [ , [ @category_id = ] category_id ]
  [ , [ @owner_login_name = ] 'login' ]
  [ , [ @notify_level_eventlog = ] eventlog_level ]
  [ , [ @notify_level_email = ] email_level ]
  [ , [ @notify_level_netsend = ] netsend_level ]
  [ , [ @notify_level_page = ] page_level ]
  [ , [ @notify_email_operator_name = ] 'email_name' ]
  [ , [ @notify_netsend_operator_name = ] 'netsend_name' ]
  [ , [ @notify_page_operator_name = ] 'page_name' ]
  [ , [ @delete_level = ] delete_level ]
  [ , [ @job_id = ] job_id OUTPUT ]
```

[ **@notify_level_eventlog =** ] *eventlog_level*

This is a value indicating when to place an entry in the Microsoft® Windows NT® application log for this job. *eventlog_level* is **int**, and can be one of the following values.

| **Value** | **Description** |
|---|---|
| **0** | Never |
| **1** | On success |
| **2** (default) | On failure |
| **3** | Always |

- sp_update_job : Changes the attributes of a job.

```
sp_update_job [@job_id =] job_id | [@job_name =] 'job_name'
  [, [@new_name =] 'new_name']
  [, [@enabled =] enabled]
  [, [@description =] 'description']
  [, [@start_step_id =] step_id]
  [, [@category_name =] 'category']
  [, [@owner_login_name =] 'login']
  [, [@notify_level_eventlog =] eventlog_level]
  [, [@notify_level_email =] email_level]
  [, [@notify_level_netsend =] netsend_level]
  [, [@notify_level_page =] page_level]
  [, [@notify_email_operator_name =] 'email_name']
  [, [@notify_netsend_operator_name =] 'netsend_operator']
  [, [@notify_page_operator_name =] 'page_operator']
  [, [@delete_level =] delete_level]
  [, [@automatic_post =] automatic_post]
```

- sp_delete_job : Deletes a job

```
sp_delete_job
  [ @job_id = ] job_id | [ @job_name = ] 'job_name'
  [ , [ @originating_server = ] 'server' ]
```

```
    [@job_id =] job_id

    Is the identification number of the job to be deleted. job_id
    is unique identifier, with a default of NULL.

    [@job_name =] 'job_name'

    Is the name of the job to be deleted. job_name is sysname, with
    a default of NULL.
```

- sp_add_jobschedule

```
sp_add_jobschedule [ @job_id = ]job_id,|[ @job_name = ] 'job_name',
  [ @name = ] 'name'
  [ , [ @enabled = ] enabled ]
  [ , [ @freq_type = ] freq_type ]
  [ , [ @freq_interval = ] freq_interval ]
  [ , [ @freq_subday_type = ] freq_subday_type ]
  [ , [ @freq_subday_interval = ] freq_subday_interval ]
  [ , [ @freq_relative_interval = ] freq_relative_interval ]
  [ , [ @freq_recurrence_factor = ] freq_recurrence_factor ]
  [ , [ @active_start_date = ] active_start_date ]
  [ , [ @active_end_date = ] active_end_date ]
  [ , [ @active_start_time = ] active_start_time ]
  [ , [ @active_end_time = ] active_end_time ]
```

[ **@freq_type =** ] *freq_type*

This is a value indicating when the job is to be executed. *freq_type* is **int**, with a default of 0, and can be one of the following values.

| **Value** | **Description** |
|---|---|
| **1** | Once |
| **4** | Daily |
| **8** | Weekly |
| **16** | Monthly |
| **32** | Monthly, relative to *freq interval* |
| **64** | Run when SQLServerAgent service starts |
| **128** | Run when the computer is idle |

[ **@freq_interval =** ] *freq_interval*

These are the days that the job is executed. *freq_interval* is **int**, with a default of 0, and depends on the value of *freq_type*.

| **Value of *freq_type*** | **Effect on *freq_interval*** |
|---|---|
| 1 (once) | *freq_interval* is unused. |
| 4 (daily) | Every *freq_interval* days. |
| 8 (weekly) | *freq_interval* is one or more of the following (combined with an **OR** logical operator):<br>1 = Sunday 2 = Monday<br>4 = Tuesday 8 = Wednesday<br>16 = Thursday 32 = Friday<br>64 = Saturday |
| 16 (monthly) | On the *freq_interval* day of the month. |
| 32 (monthly relative) | *freq_interval* is one of the following:<br>1 = Sunday 2 = Monday<br>3 = Tuesday 4 = Wednesday<br>5 = Thursday 6 = Friday<br>7 = Saturday 8 = Day<br>9 = Weekday 10 = Weekend day |
| 64 (when SQLServerAgent service starts) | *freq_interval* is unused. |
| 128 | *freq_interval* is unused. |

[ **@freq_subday_type =** ] *freq_subday_type*

This specifies the units for *freq_subday_interval. freq_subday_type* is **int**, with a default of 0, and can be one of the following values.

| Value | Description (unit) |
|---|---|
| **0x1** | At the specified time |
| **0x4** | Minutes |
| **0x8** | Hours |

[ **@freq_relative_interval =** ] *freq_relative_interval*

These are the scheduled job's occurrence of *freq_interval* in each month, if *freq_interval* is 32 (monthly relative). *freq_relative_interval* is **int**, with a default of 0, and can be one of the following values.

| Value | Description (unit) |
|---|---|
| **1** | First |
| **2** | Second |
| **4** | Third |
| **8** | Fourth |
| **16** | Last |

[ **@active_start_date =** ] *active_start_date*

This is the date on which execution of the job can begin. *active_start_date* is **int**, with a default of NULL, which indicates today's date. The date is formatted as YYYYMMDD. If *active_start_date* is not NULL, the date must be greater than or equal to 19900101.

- sp_add_jobstep : Adds a step (operation) to a job.

```
sp_add_jobstep [ @job_id = ] job_id | [ @job_name = ] 'job_name'
  [ , [ @step_id = ] step_id ]
  { , [ @step_name = ] 'step_name' }
  [ , [ @subsystem = ] 'subsystem' ]
  [ , [ @command = ] 'command' ]
  [ , [ @additional_parameters = ] 'parameters' ]
  [ , [ @cmdexec_success_code = ] code ]
  [ , [ @on_success_action = ] success_action ]
  [ , [ @on_success_step_id = ] success_step_id ]
  [ , [ @on_fail_action = ] fail_action ]
  [ , [ @on_fail_step_id = ] fail_step_id ]
  [ , [ @server = ] 'server' ]
  [ , [ @database_name = ] 'database' ]
  [ , [ @database_user_name = ] 'user' ]
  [ , [ @retry_attempts = ] retry_attempts ]
  [ , [ @retry_interval = ] retry_interval ]
  [ , [ @os_run_priority = ] run_priority ]
  [ , [ @output_file_name = ] 'file_name' ]
  [ , [ @flags = ] flags ]
```

[**@step_id =**] *step_id*]

This is the sequence identification number for the job step. Step identification numbers start at **1** and increment without gaps. If a step is inserted in the existing sequence, the sequence numbers are adjusted automatically. A value is provided if *step_id* is not specified. *step_id* is **int**, with a

```
      default of NULL.
```

- sp_delete_jobstep : Deletes a job step.
- sp_update_jobstep : Updates a job step
- sp_update_jobschedule : Updates a job's schedule.
- sp_delete_jobschedule : Deletes a job's schedule.

The equivalent of *program,* in Oracle, is called a *job* in MS SQL Server. A *job* has a schedule and steps (the third layer). The following is an algorithm for job creation :

1. Create a *job* with *sp_create_job()*.
2. Create a *job* step with *sp_create_jobstep()*.
3. If more steps proceed to step 2, otherwise continue.
4. Create a *job* schedule with *sp_create_jobschedule()*.

As one can see, the exposed interface is quite complicated. Even more so, the values passed to the stored procedures have domains of values which are quite unintuitive. In addition [MSDN] states:

*"**Remarks***

*SQL Server Enterprise Manager provides an easy, graphical way to manage jobs, and is the recommended way to create and manage the job infrastructure."*

The intention, when performing the design of the prototype presented in this thesis, is to create an easy and straightforward way for a user to create jobs (temporal triggers). Graphical tools are an extension and something good to have, but having simple low-level SQL interfaces will simplify temporal trigger administration.

## 2.3 Sybase Adaptive Server Enterprise

The basic tasks provided by AES' module "Job Scheduler" [ASIQPG] are:

- job creation, modification and deletion
- job schedule creation, modification and deletion
- scheduled job creation, modification and deletion
- job history

All entities, related to jobs creation in AES, can be administered either by using a command line tool or by using a graphical one, for instance Sybase Central.

AES imposes two security levels that are related to “Job Scheduler” (JS):

- **js_user_role** : allows creation, modification, deletion and running of jobs, but does not allow access to the underlying tables, which support JS.
- **js_admin_role**: js_user_role on all jobs disregarding the job owner. This grant also allows access to JS underlying tables.

Sybase AES exposes an interface that is similar to the one of MS SQL Server. The syntax similarity is not a coincidence, since MS SQL Server is developed from the Sybase code base. Both use T-SQL (Transact SQL) syntax, an SQL extension.

The functions exposed to the user are:

- **sp_sjobcreate**
  For creation of jobs, schedules, and scheduled jobs.
- **sp_sjobcmd**
  For managing the SQL source of a job.
- **sp_sjobmodify**
  For modification of jobs, schedules, and scheduled jobs .
- **sp_sjobdrop**
  Deletion of jobs, schedules, and scheduled jobs.
- **sp_sjobhelp**
  Gives report of all scheduled and running jobs.
- **sp_sjobcontrol**
  JS administration of scheduled and running jobs.
- **sp_sjobhistory**
  Jobs history.

Examples [ASIQPG]:

```
sp_sjobcreate @name='dev1_old_logins',
  @option='server=dev1,jname=find_old_logins,sname=daily 01:00am';

sp_sjobcreate @name='evening_sales_reports',
  @option='server=reports, jname=load_sales_data,
                 jcmd=exec    sp_new_sales_data,    starttime=23:00,
days=Monday:Wednesday:Friday';

sp_sjobdrop @name='jname=load_sales_data', @option='all';

sp_sjobcontrol @option='stop_js';

sp_sjob_control @option='start_js';

sp_sjob_control @option='stop_js_wait';
```

The smallest possible time resolution is one minute. For example, a job cannot be scheduled to be executed every 15 seconds. "Job Scheduler" can be disabled and enabled, as shown above, during run time of the server by using the *sp_sjobcontrol()* stored procedure. In addition, a running job can be interrupted during its execution.

## 2.4 Sybase Adaptive Server IQ

Sybase Adaptive Server IQ is an OLAP product of Sybase Inc., while Sybase Adaptive Server Enterprise is an OLTP product. Sybase AS IQ does not have a stored procedures interface for *jobs* administration. In the documentation of this product, jobs are known as *events*. Administration is performed by issuing SQL statements, which are not SQL standard but instead are a vendor extension [ASIQRM, page 426].

Sybase AS IQ uses a one-layered monolithic solution. The BNF is taken from [ASIQRM] and is:

```
CREATE EVENT event-name
[ TYPE event-type ]
[ WHERE trigger-condition [ AND trigger-condition ], ... ]
[ SCHEDULE schedule-spec, ... ]
[ ENABLE | DISABLE ]
[ AT { CONSOLIDATED | REMOTE | ALL } ]
[ HANDLER BEGIN schedule-statements END ]

event-type:
   BackupEnd | "Connect"  | ConnectFailed |
   DatabaseStart | DBDiskSpace | "Disconnect" |
   GlobalAutoincrement | GrowDB | GrowLog | GrowTemp |
   LogDiskSpace | "RAISERROR" | ServerIdle | TempDiskSpace

trigger-condition:
     event_condition( condition-name ) { = | < | > | != | <= | >= }
value

schedule-spec:
    [ schedule-name ]
    { START TIME start-time | BETWEEN start-time AND end-time }
    [ EVERY period { HOURS | MINUTES | SECONDS } ]
    [ ON { ( day-of-week, ... ) | ( day-of-month, ... ) } ]
    [ START DATE start-date ]

event-name | schedule-name: identifier

day-of-week : string

day-of-month | value | period : integer

start-time | end-time : time

start-date : date
```

Because of the nature of the events in Sybase AS IQ, which are more than temporal triggers, the grammar defined there is more complicated than the case where only temporal triggers have to be supported. An *event* in Sybase AS IQ is not only a job that can must be executed according to some schedule, but also situations that occur inside the RDBMS, such as connection failures, out-of-memory errors, disk full errors and so on. The DBA can create events that handle such situations. Because the aim of this master thesis is to build a working prototype only for temporal triggers, the grammar to be defined for the prototype should be not be that extensive but extendable.

Examples [ASIQRM]:

```
-- Backup event
CREATE EVENT IncrementalBackup
SCHEDULE
  START TIME  1:00AM EVERY 24 HOURS
HANDLER
BEGIN
   BACKUP DATABASE INCREMENTAL TO  backups/daily.incr
END

-- OLAP event summarizing orders on daily basis
CREATE EVENT Summarize
SCHEDULE
    START TIME '6:00 pm' ON ( 'Mon', 'Tue', 'Wed', 'Thu', 'Fri' )
HANDLER
BEGIN
    INSERT INTO dba.OrderSummary
        SELECT MAX( date_ordered ), COUNT( * ), SUM( amount )
           FROM dba.Orders
        WHERE date_ordered = current date
END
```

## 2.5 IBM DB2 Universal Database

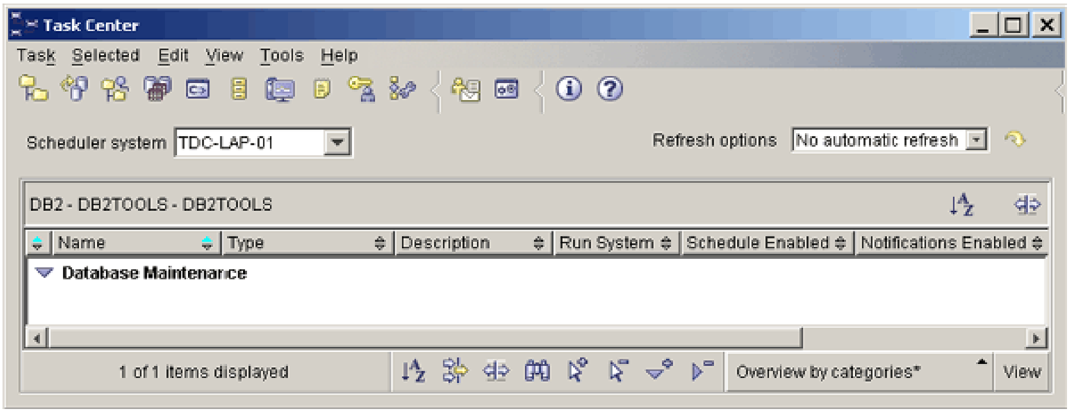
The model used by IBM DB2 Universal Database is unknown to the author of this thesis, because the documentation regarding this

functionality of the product describes only how a task, the term used inside the documentation and the applications, can be created by usage of a graphical tool [DBAC]. However, it can be concluded from the following quotation, that IBM DB2 uses a two layered structure:
"You can specify that the command runs once, or at multiple times based on a schedule or on a list of saved schedules. If you want to run multiple tasks on the same date or at the same time, it may be simpler to create one schedule record and use that for each of your tasks rather than re-creating it every time."[DBAC].

"One of the DB2 tools that is used to control the database is the Task Center. The Task Center is used to run tasks, either immediately or according to a schedule, and to notify people about the status of completed tasks.

The Task Center includes all the functionality found in the Script Center in previous versions of DB2, plus additional new features. A task is a script, together with associated success conditions, schedules, and notifications. You can create a task within the Task Center, create a script within another tool and save it to the Task Center, import an existing script, or save the options from a DB2 dialog or wizard such as the Load wizard. A script can contain DB2, SQL, or operating system commands."[DBAC].

DB2 tasks are created with DB2's Task Center GUI. After it has been launched, the DBA can create a task by selecting *New*. Seven tabs are then presented to the user: *task*, *command script*, *run properties*, *schedule*, *notification*, *task actions*, and *security*. In order to create the task, the user must identify the task type, run system, task category, and DB2 instance and partition. The type of task can be a DB2 command script, OS command script, MVS shell script, or grouping task. Under the *Command* tab the script can be imported, or entered manually. Once scheduled, DB2 can advise the administrator if jobs do not complete or if error conditions are encountered. An email can be sent every time a task has finished its execution. The email can contain arbitrary text, however only the exit code of the execution is available and not the actual results.


Task Center

Task Selected Edit View Tools Help

Scheduler system TDC-LAP-01

Refresh options No automatic refresh

DB2 - DB2TOOLS - DB2TOOLS

| Name | Type | Description | Run System | Schedule Enabled | Notifications Enabled |
|---|---|---|---|---|---|
| Database Maintenance | | | | | |

1 of 1 items displayed

Overview by categories* View

Figure 2.1. IBM DB2 Task Center ([Fierros03])

## 2.6 Summary

In this section a table that summarizes the functionality provided by the commercial systems reviewed is presented.

| | Oracle 10g | MS SQL Server | Sybase AES | Sybase AS IQ | IBM DB2 |
|---|---|---|---|---|---|
| Architecture levels | 2 | 3 | 2 | 1 | 2[1] |
| Uses SQL vendor ext. | no | no | no | yes | no |
| Uses SP interface | yes | yes | yes | no | no |
| GUI management tools | yes | yes[2] | unknown[3] | uknown[3] | yes |
| Time zone support | yes | no | no | no | no |
| Transient TT | yes | yes | yes | yes | yes |
| Reoccurring TT | yes | yes | yes | yes | yes |
| Conversion between transient and reoccurring TT | yes | yes | yes | yes | yes |
| Direct execution of binary programs | yes | yes | no | no | no |
| Date/time restriction | yes | yes | yes | partial[4] | yes |
| Result logging | yes | yes | yes | uknown[3] | yes |

# 3. Design of High Level Architecture

[1] IBM DB2 Universal Database uses a GUI client and the model is uknown to the author but there is source that may lead to the conclusion that two layerd model is used.
[2] The use of the administrative tools is recommended by Microsoft.
[3] Uknown to the author
[4] Only the starting date and time can be restricted

*"It is the tension between creativity and skepticism that has produced the stunning and unexpected findings of science."*

*Carl Sagan*

After the features evaluation of several commercial products in the previous chapter, a list of detailed requirements to the prototype to be built is presented in this one. These requirements define the high level architecture (HLA) of the prototype.

## 3.1 Detailed requirements

A detailed list of requirements with extended explanations is presented hereafter. Every requirement consists of an explanation, what should be implemented, and a supporting discussion. TT is short for temporal trigger. Event and TT will be used interchangeably.

These requirements, defining the HLA, are based on the extended review of the commercial systems presented in Chapter 2, as well as on two talks between the author of this thesis, Mr. Peter Gulutzan and Mr. Sergei Golubchik. Mr. Gulutzan is a Senior Software Architect at MySQL AB and author of [SQL99Compl]. Mr. Golubchik is a Senior Software Engineer at MySQL AB and the technical supervisor of this thesis.

- **Types**: The event type, a required value that is either transient or recurring. An event is transient when it is executed once at one specific moment in time. A recurring event is scheduled for execution more than once. A recurring event can be executed only once, depending on its parameters, but this does not make it a transient event. For recurring events a new keyword *EVERY* must be introduced. Thus, an event can be executed *EVERY* expression *INTERVAL_TYPE* (for example EVERY 5 MINUTE). MySQL SQL syntax supports different types of intervals like minute, hour, day, and so on. The construction must be extendable. When defining a transient event the *AT* keyword must be used and introduced.

  **Discussion**:

  As a parallel, the Unix program *crontab,* and its daemon *crond*, are used for scheduling reoccurring events while the *at* program is used for transient execution of commands. However, *crontab* does not support time precision up to seconds but minutes. Hence, it is not possible to schedule an event for execution every 5 seconds, which might be desired in a particular scenario.

All RBDMS products discussed in Chapter 2 permit the creation of both transient and recurring temporal triggers.

- **Type conversion**: It should be possible to transform a transient event into reoccurring event and vice versa.

  **Discussion**:

  A *cron* job can be scheduled to be executed at specified time but *at* does not allow that, since its purpose is to execute commands only once. All systems, reviewed in Chapter 2, allow modification of the execution plan and thus allow type conversion.

- **Execution discipline**: The execution of the events must be parallel, and for every event a separate thread must be spawned. However, a specific event will be executed in serialized manner; namely no new thread will be started, when an event runs so long that it passes the time for its next execution. Therefore, if a reoccurring event takes more time to execute than the time between 2 executions, a FIFO discipline will be used for the execution.

  **Discussion**:

  The *crontab* program spawns every command in separate process like *at* does. However, *crontab* will not make a FIFO queue if a previous started command has not finished and must be executed again.

  All systems, reviewed in Chapter 2, implement parallel execution of events. However, the execution of a single job is not in parralel in Oracle 10g. Even more, the time for the next execution is calculated after the current execution has finished; this serializes the execution of a single job and still having jobs executed in parallel.

  It was chosen temporal triggers to implement execution of single entity in serialized manner, because in cases of long running processes this may lead to a heavy load of the server, which is probably unwanted behavior.

- **Termination**: Every event must be stoppable during its execution by using the SQL statement *KILL*. A process number has to be provided, which can be found from the information returned by *SHOW PROCESSLIST* or "mysqladmin processlist" at the command prompt.
  **Discussion**:
  As mentioned above, the commands executed by *crontab/at* are started as OS processes and thus can be killed under Unix by using the *kill* command, and on other operating systems using appropriate tools. All products reviewed in Chapter 2 allow the user to stop currently running temporal trigger by means of a graphical or command-line tool.

- **Ownership**: An event has a definer and the name of this user has to be stored within the event's metadata. Later the event must be executed with the rights of the user who has defined the temporal trigger. One user should not be able to create events that are executed with the rights of another user.
  **Discussion**:
  Sybase AS IQ documentation reads: *"The event name is an identifier. An event has a creator, which is the user creating the event, and the event handler executes with the permissions of that creator. This is the same as stored procedure execution. You cannot create events owned by other users."* As you can see, Sybase AS IQ uses the same policy. Moreover, no known system, that provides temporal triggers, permits events to be created by one user and executed with the rights of another.
  The *crontab* utility creates jobs to be executed per user and the metadata regarding execution plan as well as the command to be executed are stored in a separate file per user. Hence, the task is executed with the user's rights. However, a superuser may create a

task for another user by using "-u" command line switch of *crontab*.

- **Commands executed by temporal triggers**: A temporal trigger always has an associated anonymous SQL block. Execution of external entities like shell scripts and binary programs can be performed by calling an UDF in the SQL block, or in a stored procedure (SP) called from the anonymous SQL block. A User Defined Function is a routine written in C/C++ or another language, which binds by using C/C++ function calls and the stack. UDFs are loadable by the MySQL server at run-time. A UDF resides in a DLL, when the OS is Windows, or in a SO (shared object) on most Unix flavors.

  **Discussion**:

  Commands executed by *crontab/at* can be quite complex, and are bound by the limitations of the command interpreter used by *crontab/at* for executing. The complexity on other OSes is different.

  MS SQL Server allows the defining of job steps, hence allowing execution of high complexity jobs.

  All other systems, reviewed in Chapter 2, do not use job steps but a single entity to be executed. The complexity in all cases is limited to the complexity of the run-time (interpreter/compiler) engine of the product.

  The reason for not adding the possibility temporal triggers to execute binary and shell programs is that this is not the right module to right this functionality. Havind an user defined function (UDF), which does that is a better solution, since then also MySQL's stored procedures and table triggers can benefit from it.

- **Output logging**: Any output is saved into a log file, which is in human readable text form. The name is specified at the MySQL server startup or in the server configuration file (section [mysqld]). The command line option is named *--evex-log* and the option in the server configuration file is named *evex-log*.

  **Discussion**:

  If the output of the command executed by *crontab* is not redirected to a file (in most cases /dev/null) then all the output generated is sent, using the email service on the machine, to the user that has defined the cron job.

Oracle 10g logs execution status and this information is easily traceable by using a view *DBA_SCHEDULER_JOB_RUN_DETAILS*.

Sybase AES logs the output of an execution, as well as a log of what events has been executed, internally. Stored procedures can be used to view this information.

IBM DB2 provides, through its graphical administration tool Control Center, the ability to review the exit status of executed tasks.

MS SQL Server can log the output to the system logger of Windows NT in case of either success or failure, or both. This is configurable.

The *--abc* type of command line options and the use of config files are standard MySQL feature and thus consistency with the established rules is desired; these are found intuitive by the users of MySQL.

- **Time zones**: When defining an event the current (connection's) time zone must be used, however the storage of datetime must be according to UTC (Universal Coordinated Time). Because MySQL's datetime syntax does not permit specification of a time zone (for example "*2004-12-26 15:00:00 CET*") the current time zone must be used. An attempt to use the value presented above will create a truncation error inside MySQL and generate a warning.

  As it will be explained in details in Chapter 4, there is a time zone, named SYSTEM, in MySQL which is the server's time zone. It can be changed during startup or otherwise the operating systems setting for time zone is used. When a new thread is created, to handle a connection, this thread inherits the current SYSTEM time zone from the the global server settings. However, it is possible an user to change the this setting on connection level without affecting the globals setting.

  **Discussion**:

  There is a specific reason to choose UTC as the time zone. Using UTC to store the time when a temporal trigger fires, removes some side effects from the nature of daylight saving time (DST).

Since UTC never has daylight savings, a specific moment occurs only once. On the contrary, if the time is stored in the SYSTEM time zone, times like 02:23 occur twice on a day in November when the clock is “moved back” (See the explanation of DST in Glossary).

As an example let's assume that SYSTEM time zone is CET (Central European Time), which is defined as UTC+1 and during daylight savings time as UTC+2. The clock is moved back at 03:00 on a specific date in November (see DST in Appendix A). At 03:00, before moving the clock back the UTC time is 02:00, and when moving it back the UTC time is still 02:00, since the daylight saving time is no longer in effect (CET+1). Therefore, while a wall clock will show 02:23 two times in one day according to CET, the UTC wall clock will never show 02:23 two times on the same day.

*crontab* and *at* use the computer clock and does not consider time zones. Therefore, if the OS is set to a time zone that has daylight savings, the problem with double execution or no execution (when the clocks are moved forward in March every year) may occur.

Oracle 10g supports time zones as part of a datetime value, thus the connection's SYSTEM time zone (TZ) is not used but the one specified. If no TZ is specified, the current system TZ is used.

MS SQL Server does not allow TZ as part of *active_start_time* and *active_end_time*.

Sybase AES is similar to MS SQL Server and does not allow TZ as part of *starttime* and *endtime* whenever a schedule is defined.

MySQL will add, in the future, time zone specification as integral part of a datetime value. When this is done temporal triggers can be scheduled with time zone in mind.

- **Execution plan time restrictions**: It should be possible to set the interval for a recurring event. For this, the *STARTS* and *ENDS* keywords need to be introduced in the grammar. After both of these keywords, a datetime value, or an expression that evaluates to datetime value, should be specified. The server must check for input data validity whenever applicable:

`STARTS <= datetime <= ENDS or STARTS <= datetime_expr <= ENDS` *STARTS* can be in the past compared to the value returned by the function NOW(), which returns the current datetime as a Unix timestamp, according to the connection's time zone.

An additional clause (*ON COMPLETION [NOT] PRESERVE*) determines whether the event will be automatically deleted (dropped) or preserved, after which it will not available to execute. This happens when *ENDS* becomes a datetime in the past. The default behavior is to drop the event. The clause *ON COMPLETION PRESERVE* is also applicable to transient (one-time) temporal triggers.

**Discussion**:

The *at* command "forgets" about what was executed just after the command has been started. *crontab* does not support limiting the time interval for a command to be executed, but this can be worked around by using the fairly complex syntax which specifies when the command will be executed.

"The Scheduler", of Oracle 10g, and its interface permit limitation of the time period, by means of *start_time* and *end_time* arguments to the *dbms_scheduler.create_job()* stored procedure. A job is defined as completed when the execution plan does not allow any further execution. In this case the job is deleted. The DBA does not have the choice to preserve the job as disabled.

MS SQL Server allows execution time restriction when the schedule plan of a job is being defined. Start and end time, as well as start and end dates, are separate entities. This is also valid for Sybase AES. Sybase Adaptive Server IQ only allows definition of a start date and start time. If they are not provided, they default to current date and time, which is the behavior of all systems reviewed in Chapter 2.

- **Privileges**: For creation, redefinition and deletion of events a new privilege level *EVENT* has to be introduced.

**Discussion**:

TTs are separate database objects; therefore there is a need to be able to restrict their usage. Moreover, TT in the hands of non-experienced or malicious users could lead to system performance problems.

The need for a new privilege comes from environments with many users, for instance web/database hosting environments, where the DBA should be able to restrict the usage of TT on per user level. On the other hand, there is no reason to introduce more than one privilege, because if one has the ability of creating events as database objects therefore he must be allowed to change them. This is on the contrary with the privileges on table level where there are SELECT, INSERT, ALTER and other privileges. Adding more than one privilege will make things more complex.

- **Metadata storage**: Event metadata must be stored in catalog *mysql*. When creating an event, the dot notation (*database.sp_name*) must be used in the anonymous SQL block, whenever a stored procedure from another catalog is referenced. If this is not the case then a short name is sufficient. If a short name (without the schema specified) is used, then the event is created as an event of the current database. A database must have been selected before creating a temporal trigger.

  **Discussion**:

  All systems, reviewed in Chapter 2, store the metadata in a place which is usually invisible to the normal user. The only way to change and query this metadata is to use an already defined interface. In the case of Stored Procedures, introduced in MySQL 5.0.0, the metadata is stored in the *mysql* catalog. On the contrary, MySQL triggers (also introduced in the same version), are stored in an external file with extension *.trg* . As a matter of fact, the table definition and its metadata, is stored in a file with extension .frm. Nonetheless, trigger implementation is subject to change. Hence, the metadata will be moved to the .frm file.

  Trigger definitions are not stored the *mysql* catalog but accompany the .frm file. The reason for not being stored in the *mysql* catalog, is that they are integral part of the table definition, while temporal triggers are not table but system specific, like for example stored procedures are. Hence, one may conclude that temporal triggers must be stored in the *mysql* catalog, in a separate table.

- **<u>Validation</u>**: When a TT uses a stored procedure that has been dropped, the former will be executed and the result will be an error. However when the TT is defined no check is made whether the SP or the procedures, if many are used, exist. Wherefore, the anonymous SQL block is checked only for syntax validity (*linting*), when the temporal trigger is defined.
  **<u>Discussion</u>**:
  *crontab* and *at* utilities does not check the semantic validity of the command lines to be executed. Even more, the behavior is similar to the one of stored procedures, which are interpreted at run-time and only syntactically checked during definition. Stored procedures are stored compiled only in RAM but never on the hard disk in this state (see 4.8).

- **<u>Module administration</u>**: The Event Executor (the module that executes declared events), or in short EVEX, can be disabled at database startup with a command line switch. Its name is *--event-executor* and possible values are 0 or 1. The Event Executor can be disabled in MySQL's configuration file, in the *[mysqld]* section. At runtime the behavior can be controlled by a global scope server variable, namely event_executor:
  ```
  SET @@event_executor = 0; -- disable EVEX
  SET @@event_executor = 1; -- enable EVEX
  ```
  This could be useful for temporarily disabling the execution of events. The reasons could be different, for instance high load of the database server or a backup procedure being executed by the database administrator, which should not be interfered with any way. In addition, the command “FLUSH EVENTS” must be implemented, which forces the EVEX to throw out all information cached in memory and recreate its in-memory structures by re-reading all needed data from *mysql* schema. For flushing, the user has to have the SUPER privilege. Finally, a compile time option must be available to disable compilation of the

module and subsequent object linking into the server binary. The name of the command line option passed to the configuration script is: `--disable-event-executor`.

**Discussion**:

Different modules of the RDBMS can be opted in and out during the pre-compilation phase, the so called *configure* phase. MySQL uses Makefiles for compilation and on most platforms the *autoconf* and *automake* tools. The first is used to create a independent system for compilation configuration that is not OS specific and creates makefiles which are used by the *automake* tools to compile the sources.

In addition, there are a handful of FLUSH commands already implemented and used by the MySQL replication module or by the core of the server.

- **Replication**: The SQL statements executed by events should not be replicateable by the MySQL replication mechanism.

  **Discussion**: Replicating SQL statements imposes severe problems:

  1. All SQL statements that change data have to be excluded from the binary log used by the replication mechanism. If the events are being replicated as well as the SQL statements issued during their execution, the events in the replicas have to have the logging of their SQL statements disabled. However, this is a circle with no exit, since there should be no way to distinguish a master from a replica. Even more so, in case of failure the logging of the SQL statements has to be somehow re-enabled. In event of failure of the master, the fail-over plan redirects all connections to the slave, the slave has to work as if it is the master.

     It is possible that the slave executes the events not in the same order they were executed by the master. Therefore, it might happen that the slave is not exact replica of the master. This is a counterpoint to the implication that the SQL statements should not be replicated.

2. The situation gets more complicated when the replication structure is complex, for instance multi master or tree. In case of multi-master layout of the network execution of some events, like ones that perform summarization, may not work. On the other hand, others will work, such as management of session data, where an event has to delete all stale sessions after some time.

- **Run-time statistics**: When an event is created or executed, MySQL server statistics have to be updated; namely new statistical variables have to be introduced that measure the number of created/altered/dropped events as well as the number of executed events so far, counted from server startup.

  **Discussion**:

  MySQL collects all kind of statistics about the way it performs. Having more available gives more information to the DBA.

- **Backup and export**: A clause like *IF NOT EXISTS,* when defining, and *IF EXISTS,* when dropping, must be allowed to prevent a SQL statement from failing. When the former clause is used and an event with the same fully qualified name exists, a warning must be generated, but the SQL statement must not fail. The same is valid whenever a user tries to drop a non-existing event. The warnings are available to be seen by executing the *SHOW WARNINGS* command immediately after the statement (import) has been executed.

  **Discussion**:

  The grammar of MySQL's DDL allows the use of both *IF NOT EXISTS* and *IF EXISTS,* which are used to prevent the execution of a statement from failure but instead generate a warning. For instance:

  ```
  CREATE TABLE IF NOT EXISTS some_table (....);
  ```

  These error preventing clauses are used to ease the process of restoring data from backups.

  Adding support for these clauses makes the temporal triggers' grammar consistent with the MySQL SQL grammar.

- **Multithreading**: Whenever a new thread is spawned for an event to be executed, the MySQL server's thread descriptors cache must be used to reduce the overhead of creating a new thread handle, in case a cached (no longer used but not destroyed) one exists.
  **Discussion**:
  MySQL internally has a pool of thread handles ready for re-use to lower the overhead of creating a new one every time a new thread is needed. The idea is similar to the database connection pooling used in many programs.

- **Self referencing**: An event should be allowed to drop itself in its SQL block. In this case the event should be removed (uncached) from EVEX in-memory structures.
  **Discussion**:
  An event must be allowed to drop itself, however there are other ways of dropping an event, for instance by using the ON COMPLETION NOT PRESERVE clause. However, stored procedures in MySQL are not allowed to execute statements related to stored procedures.

- **Testing**: During the development of the prototype, the "Test First" approach defined by the eXtreme Programming [Beck04] process must be used.
  **Discussion**:
  This approach is also known as "Test-driven development". It is used in all "agile" software development processes. It is known that the test-driven development leads to less defects per thousands lines of code.

## 3.2 Temporal triggers SQL grammar

The grammar is presented in EBNF (ISO/IEC 14977:1996(E))[EBNF]. EBNF stands for Extended BNF. EBNF extends BNF with regular expression suppport which makes the definitions compact. "While a BNF notation can be specified in a few sentences, the proper definition of EBNF requires a little bit more explanation, and therefore frequently only BNF is used although the result is much less readable."[EBNF2]. EBNF is a bit different than the normally used BNF. A list of differences that are related to the grammar follows:

- terminals which are strings are presented quoted
- non-terminals may have a space in the name
- elements are concatenated with comma ","; comma is a separator.

```
(* creates an event *)
create_event =
         "CREATE EVENT", event_name , "ON SCHEDULE" ,
         schedule_time
        [, "ON COMPLETION" [,"NOT"] ,"PRESERVE" ]
        [, "DISABLE" | "ENABLE"]
        [, "COMMENT", comment]
        , DO , sql_statement

alter_event =
(* changes properties of an event *)
        "ALTER EVENT",
        event_name
        [, "ON  SCHEDULE ", schedule_time ]
        [, "RENAME TO", event_name]
        [, "ON COMPLETION" [,"NOT"] ,"PRESERVE" ]
        [, "COMMENT", comment]
        [, "DISABLE" | "ENABLE"]
        [, DO, sql_statement ]

(* use CALL sp_name(par1 [, ...]) to call SP (according  to the SQL
standard). Statement can also be a compound statement surrounded by
BEGIN and END keywords *)

(* deletes an event*)
drop_event = "DROP EVENT", event_name

(* shows how an event will be defined in SQL*)
show_create_event = "SHOW CREATE EVENT", event_name

(*shows a list of all events with detailed information *)
show_events_status =  "SHOW EVENT STATUS"

(*granting/revoking of a priv. for create/alter/drop/exec event *)
grant_priv = {"GRANT" | "REVOKE "},  "EVENT"   .....


(* flushing :re-reading information from the mysql schema *)
flush_events = "FLUSH EVENTS"

schedule_time = "AT" datetime
       | "EVERY", expr , interval_expr (*as in sql_yacc.yy *)
        [ "STARTS", datetime ,]
        [ “ENDS”, datetime    ]
 (* expr non-terminal is a valid expression that evaluates to a scalar
value whose integer part is used later. *)

event_name = follows the way of naming tables in MySQL([MySQLRef]
section 10.2)
```

## 3.3 Summary

The BNF presented in 3.2 shows syntax that is similar to the syntax defined in SQL-92 for different kinds of RDBMS objects. The intention was to use SQL for temporal trigger administration instead of building an interface of stored procedures that do the administration behind the scenes.

The grammar is extendable because in the future handlers for non-time based events might be needed. Because of that temporal triggers are created with the `ON SCHEDULE` clause. If the grammar has to be extended an example for creating an event that handles disk full error might be :

```
CREATE EVENT event-name ON EDISKFULL ...
```

Hence, `SCHEDULE` is just a type of event.

It must be noted that the author of this thesis reviewed Sybase AS IQ, which has very similar BNF to the one presented above, after he defined the grammar to be implemented in the prototype. Therefore the BNF presented in this thesis and the BNF of Sybase AS IQ should be considered independent works.

A short list of requirements follows:

1. Implementation of transient and reoccurring temporal triggers.
2. Convertibility of transient into reoccurring temporal trigger and vice versa.
3. Parallel execution of temporal triggers. Sequential executions of a particular temporal trigger.
4. Possibility to terminate a running temporal trigger.
5. Every event must have an owner. The owner is the event definer.
6. The body of a temporal trigger should almost use the grammar used for stored procedures.
7. The result of every execution has to be logged into a text file.

8. Start/End time restrictions must be implemented.
9. Temporal triggers should be aware of time zones and DST.
10. New privilege, namely EVENT, has to be implemented.
11. Temporal triggers must be stored in the *mysql* catalog.
12. When an event is defined its body must be checked only for syntactical validity. All other checks are performed when the temporal trigger is executed.
13. Execution of TTs must be controllable by a global server variable. This variable activates and deactivates the events executor module, namely EVEX, during run-time.
14. Conditional linking into the server's code.
15. Statistical variables.
16. *IF EXISTS* and *IF NOT EXISTS* clauses.

# 4. Internals of MySQL RDBMS related to the implementation of temporal triggers

*If you have knowledge, let others light their candles in it.*

*Margaret Fuller*

After presenting the requirements of the temporal triggers prototype to be built in the previous chapter, in this chapter we continue with an explanation about the inner workings of the MySQL RBDMS, which we relate and correlate to the idea of temporal triggers.

The explanation starts from more broad terms and their behavior and implementation and moves at the end of the chapter to quite specific ones like stored procedures and triggers in MySQL.

## 4.1 Client/Server

MySQL RDBMS uses client/server as an architecture. A version of MySQL exists; namely libmysqld, also known as embedded server, which does not use this architecture. However, this version is not related to this thesis, since temporal triggers have less value in such a scenario.

"Client/Server is a network application architecture which separates the client (usually the graphical user interface) from the server. Each instance of the client software connects to a server or application server.

Client/Server is a scalable architecture whereby each computer or process on the network is either a client or a server. Server software generally but not always runs on powerful computers dedicated for exclusive use to running the business application. Client software on the other hand generally runs on common PCs or workstations. Clients get all or most of their information and rely on the application server for things such as configuration files, stock quotes, business application programs or to offload compute intensive application tasks back the server to keep the client computer (and client computer user) free to perform other task."[WikiCS].

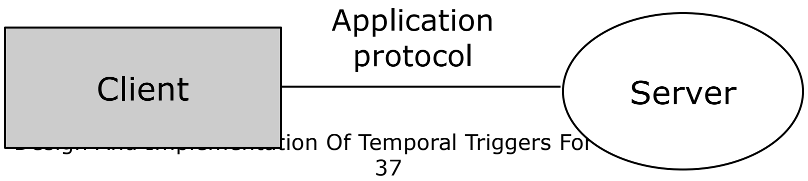


Figure 4.1 Client/Server architecture

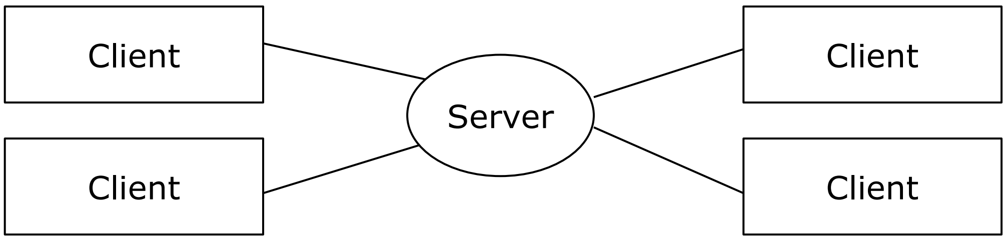


Figure 4.2 Server communicating with many clients at the same time

The MySQL server allows different kinds of connections. The following mechanisms are used:

- Unix domain sockets (Unix and Unix-like only)
- TCP sockets
- Named pipes (Windows OS only)
- Shared memory (Windows OS only)

The Unix domain sockets are not designed for client/server communication in a network but are a way of performing client/server communication on a single host. The Unix domain sockets can bee seen as an alternative to the interprocess communication (IPC) methods; namely shared memory, semaphores and message queues. The reason for being an alternative of the latter is that the client and the server are on the same host. Unix domain sockets are two types:

- stream sockets (similar to TCP)
- datagram sockets (similar to UDP)

The Unix domain sockets are available only on Unix and Unix-like operating systems, which is the reason to have "Unix" in the name. The identifier of a Unix domain socket is a file name, including path name, on the local file system. However, this file is not an ordinary file from which one can read and write data into, by using standard library functions; to do so one has to use a set of other functions. The reason these are preferred, when the communication is between processes running on the same host, is because Unix domain sockets provide faster communication than TCP sockets. Shared memory outperforms Unix domain sockets, but the interface of the former is not built for writing network applications.

TCP (Transmission Control Protocol) sockets are now the generic way for communication between programs running on different hosts in a network (LAN, MAN, WAN). The underlying protocol in the OSI (seven layered model) model is IP and this is the reason TCP is often written together with IP as TCP/IP. TCP is a stateful protocol, which guarantees ordered delivery of data without errors. TCP/IP enables computers with different architectures to "talk" to each other. In addition, TCP is the de facto standard these days for reliable communication.

Named pipes are an extended version of pipes, known in the computer science literature as FIFO queues. FIFO is an acronym showing data flow; first in, first out. The other discipline is LIFO (last in, first out), which is used for stack implementations. Named pipes, as way of communication between a client and the MySQL server, are only available on Windows OS. Their performance is worse than that of TCP sockets, but they have the advantage of being only accessible from the local host; therefore a malicious attack from outside is impossible (the same applies for Unix domain sockets). This is different than TCP sockets, which are accessible from other machines in a network.

Shared memory is a way of communication between processes (IPC – interprocess communication). When processes use shared memory, they ask the OS to give them access to a specific shared memory segment. The operating systems maps addresses from their virtual memory space to a region in the memory where the shared data resides. When performing reads and writes to the mapped segments the kernel of the OS remaps and accesses the real memory location. RAM (Random Access Memory) is a 1st level memory, which has fast access and write timings, and is *n* times faster than any secondary memory type, such as hard disk. This makes shared memory a very efficient method of communication within a single system. The security advantage of named pipes and Unix domain sockets is also present.

Because of all these different types of communication channels that are supported by the server, there is an abstraction layer, namely VIO, which isolates the communication from the specifics of the medium. VIO stands for Virtual Input Output. Adding another kind of medium imposes only creation of an implementation of the VIO interface for it. VIO is located in the *vio/* subdirectory of the MySQL server source tree. It is implemented in C, instead of C++, because the client library, namely libmysql and libmysql_r (the one to be used by multithreaded applications) is implemented only in C for better portability (not every platform has good C++ compiler, but a C compiler is always available).

## 4.2 Processes and Multithreading

An operating system process, or just process, is an entity residing in memory that has its own stack, heap and code segment. The stack and the heap are sometimes known as data segments. A process is not a program. The latter is a file on disk.

Processes are scheduled for execution by the OS and are always in one of (usually) 3 states: ready, running or blocked. For every state there is an in-memory structure that holds descriptors of all processes in the state. This structure resides in the kernel memory space. The structure used depends on the discipline used by the OS scheduler to schedule processes for execution. It may be just a simple linked list, in the simplest case, or a heap sorted with heap sort. The heap structure is very convenient and heap sort is a fast algorithm, which has guaranteed speed of O(n.log(n)) [AlgHeap][DADS]. The heap is an array (sequential block of memory) that is used in a non-sequential manner. Item insertion is a very fast operation, and depending on the direction of the sort, the element with the smallest or greatest value is in the beginning of the memory block. Hence, the scheduler always has to fetch the first element.

The contemporary operating systems isolate processes and provide specifics interface for intercommunication. One process cannot read and write to the memory of another. If a process crashes, for some reason, this will not lead to instability of the system as a whole, but only the process will be terminated and its memory, as well as all other used resources, will be freed.

Still, there are situations where such constraints are too restrictive and have to be loosened. Usually the memory isolation is one of the unwanted constraints and once loosened this eases the inter-process communication. For such reasons, a new entity named *thread* exists. Threads live inside a single process. Threads are sometimes called light-weight processes (LWP). The number of threads per process is limited only by the internal structures of the operating system, when the kernel is thread-aware, or the thread library structures, when the threads are implemented with a library. Both approaches have pros and cons.

When a library is used, the kernel memory space does not grow a lot and the context switching is fast, because it does not include a system call (kernel call), which switches the processor to execution in Ring 0 from execution on Ring 3 (kernel and processes run on different privilege rings). Still, if a single thread in a process blocks for some reason, for instance I/O operation blocked, then the whole process is blocked, because the kernel does not know anything about other threads in the process. Hence, no other thread will have the chance to be executed until the thread that has been blocked unblocks.

This does not happen when the kernel is aware of threads. The OS will not block the whole process but a switch between user mode and kernel mode is expensive.

MySQL server is a multithreaded program. The reason behind the choice to use threads instead of processes that communicate between each other by using IPC(inter process Communication) primitives is that the use of the process memory is faster than the use of the shared memory. In both cases the access to memory is guarded by using mutexes, also known as binary semaphores, invented by E. Dijkstra. Mutex stands for Mutual Exclusion. The code flow that holds the mutex prevents other code flows from acquiring it and thus prevents them from executing specific code sections, which cannot be executed in parallel. Such code sections are known as critical sections. The access to shared memory usually should be synchronized with a mutex.

This implementation is used on Unix and Unix-like operating systems and other operating systems that use a POSIX implementation of threads (aka pthreads). On other platforms, like Windows OS or OS/2, an emulation layer is used. In fact, there is a thin emulation layer in the server's code (*include/my_pthread.h*), which uses C preprocessor commands to translate the common API to the available underlying functions provided by the OS. In addition, by using the preprocessor, some names of standard POSIX thread routines are replaced with the names of equivalent extended functions, which eases the usage of POSIX threads. For example, functions that acquire and release mutexes are overridden to throw an error in error conditions, like double release of an mutex or usage of an uninitialized mutex.

For every connection to the MySQL server a new thread is started, which handles it from the start, when negotiation according to the application protocol takes place, to the end, when the connection is closed by one of the parties. Every thread has its own stack and descriptor, which instance *class THD* (*sql/sql_class.h*) (THread Descriptor). The thread descriptor holds all needed connection data, like open tables, pointers to blocks of preallocated memory blocks, the connection time zone, connection level variables, connection's default character set (charset), pointer to the SQL statements parser and others.

The main MySQL thread is started when the server is started. This is the thread that awaits connections and spawns new threads for every incoming connection. Before the moment when the server is able to accept connections from the clients a series of initializations take place. They are (listed in the order they happen):

1. my_sys library (described in 4.4) and pthreads' most important mutexes are initialized. These mutexes are:
    - THR_LOCK_malloc
    - THR_LOCK_open
    - THR_LOCK_lock
    - THR_LOCK_isam

- THR_LOCK_myisam
- THR_LOCK_heap
- THR_LOCK_net
- THR_LOCK_charset
- LOCK_localtime_r
- LOCK_gethostbyname_r
- THR_LOCK_lock

2. SYSTEM time zone initialization
3. The binary log, normal log and slow log are initialized (POSIX mutexes and conditions are initialized).
4. Values for different parameters that control how the server performs are loaded from configuration file (usually /etc/my.cnf on Unix or c:\windows\my.ini on Windows OS) (*mysys/default.c::load_defaults()*). All variables are loaded into *argv*(the array with parameters every C/C++ program has in its declaration). After *argv* is parsed to set values for server parameters, like where is the data directory or on which port the server should listen (*sql/mysqld.cc::get_options()*, which invokes *mysys/my_getopt.c::handle_options()*).
5. Global and per connection system variables are initialized (*sql/set_var.c::set_var_init()*).
6. Initializing default server charset and default server collation (the rules used when strings are compared).
7. POSIX signals are initialized. Signal handlers are installed (*sql/mysqld.cc::init_signals()*). MySQL server should react on signals like SIGHUP (and restart itself) and SIGSEGV (Segmentation Fault, equivalent of General Protection Failure on Windows OS).
8. Thread stack size is changed to fit the expected needs.
9. SSL is initialized. Since 4.1.0 MySQL server provides means of encrypting connections by using Secure Sockets Layer protocol.
10. Networking (TCP sockets) is disabled if the server was started with the appropriate option –*skip-networking*.
11. If networking is enabled the server binds to its port (by default 3306). If Unix domain sockets are enabled one is created and opened.

12. The following server modules are initialized in (*sql/mysqld.cc::init_server_components()*)(listed in order of initialization):
    - the table cache (*table_cache_init()*)
    - the hostname cache (host names are not resolved every time a resolution is needed but a cache is used).
    - the query cache (since MySQL 4.0.0 a query cache exists which caches results from queries, thus speeding up the execution of regularly executed queries).
    - the random number generator
    - if replication is enabled the replicas (slaves) array.
    - a handful of logs are opened: binary, slow , error, and InnoDB logs.
    - the database handler abstraction layer is initialized.
    - if wanted and possible, MySQL is locked into memory. The OS must not swap its code.
    - the full text search engine is initialized (the stop words).
    - the connections array
13. ACL (Access Control Lists) module, which is used for checking of privileges.
14. If the server instance is a replica of a server then slave initializations are performed and slave threads are started. The replication mechanism implemented in MySQL does not transfer images between the master and the replicas, but the queries that manipulate data (DDL and UPDATE/DELETE/INSERT statements) and affect rows (an UPDATE statement may affect 0 rows and thus it is not logged) are transferred and executed on the slave's side, even if it relies on a specific per server value like one from the random number generator. Since 4.0.0, the replication mechanism uses 2 threads: a network and a SQL thread. The network thread fetches deltas, the queries executed on the master since last fetch from the replica, and stores them in a log file, named the relay log. The SQL thread fetches queries from the relay log and executes them. By using two separate threads the relay log is

almost always up-to-date with the master and the query execution time does not affect how many queries the replica is behind the master. Till 4.0.0 there was only one thread, which used to fetch and execute in sequential manner. However, if a query took long to execute and during this time the master failed, the queries executed by the master since the last fetch are lost. In case of fail-over, the replica will be behind the master, sometimes significantly.

## 4.3 Memory management

The management of memory is a very important issue during application development. C and C++ are languages that provide full access to the hardware. This can be a powerful tool in the hands of experienced programmers or a disaster in the hands of inexperienced users.

In C/C++ the memory management is in the hands of the programmer. Everyone must remember to deallocate the memory previously allocated. There are situations where multiple objects are created and keeping track of them is not an easy job. In Java, and other "new" languages, the memory management is automatic. If an object (or memory) is not referenced any more it will be automatically deleted. There are some products which bring the garbage collection to the C++ world. However, garbage collection has deficiencies in terms of speed. The garbage collector should be well tuned and quite intelligent, as is the case with the JVM (Java Virtual Machine) garbage collector, which evolves with every version.

MySQL server does not use garbage collection but a different way of keeping track of allocated memory and minimizing memory leaks because of non-freed non-referenced memory. In many cases the standard *::new* operator (the global *new operator*) is overridden to use MySQL's own memory allocator.

The server's code is aware of this allocation mechanism to some extent. The idea is that allocation of memory is slow, therefore calls to *malloc()* and similar should be used as little as possible. Moreover, keeping track of thousands of objects is not an easy job, but it can be eased by using memory pools (or memory arenas).

A memory pool is a structure which holds pointers to preallocated free memory (organized as linked list of blocks), different statistical variables and linked lists of memory for reuse (optional). Whenever memory is needed it is allocated from a block of memory that was allocated when the memory pool was created. When there are no free blocks a new block is preallocated and memory is allocated to the user from there. The memory pool structure keeps track of the last allocated byte in every block to be able to know how much free memory is left there.

When using a memory pool the server code does not need to deallocate the memory it uses. The *delete* operator must not be called but if called it will not perform anything. All allocated memory can be freed with one function call and passing the memory pool as parameter.

Memory pool, which is struct MEM_ROOT, is used in several parts of the server code like:

- query parsing (after the query has been parsed and executed the memory pool thd->mem_root is deallocated)
- stored procedures – SPs in MySQL use their own memory root for allocating memory. It is used during compilation of a SP block (sp_proc_stmt non-terminal symbol in the grammar, see 5.1).

*struct MEM_ROOT* as well as all functions, which are used for memory management through memory pools, are declared in *mysys/my_alloc*. A pool is initialized with a call to *init_alloc_root()* and released with *free_root()*. The latter call makes the pool unusable. On the other hand, it is possible to free all the allocated memory, without destructing the memory pool, by calling *mark_blocks_free()*. For instance, there is a function *strdup_root()* (*mysys/my_alloc.c*) which creates a copy of a string and uses a memory pool for memory allocation of the copy.

Classes and respectively objects, which need to have themselves stored in a memory pool, have to inherit from *class Sql_alloc*. This class overrides the standard *::new operator* with a version which allocates memory from a pool with by calling *gptr sql_alloc(uint Size)*(*sql/thr_malloc.c*). The latter turn calls *alloc_root()* and passes as a parameter a pointer to the memory pool of the current thread. *thd->mem_root* can be extracted from the thread because thd is stored as thread specific data (POSIX Threads functionality) [POSIXThr, chapter 5.4]. The way it is done is :

```
MEM_ROOT *root= *my_pthread_getspecific_ptr(MEM_ROOT**,THR_MALLOC);
```

*MEM_ROOT*** is used by the macros *my_pthread_getspecific_ptr()* (a wrapper of *pthread_getspecific()* on Unix and Unix-like, on Windows OS wrapper of *TlsGetValue()* ) to cast the returned variable since it is stored as *void* *. THR_MALLOC is the key used to retrieve it, like from a hash.

*operator delete* of *class Sql_alloc* is implemented because this is the recommended practice whenever *operator new* is overridden. Still, *operator delete* does nothing because the real deallocation is made by calling *free_root()* as mentioned above.

### 4.4 Query processing

A query is usually processed when it comes from a client to be executed. However, there is at least one case when a query does not come from a client, for instance in Trigger code.

The processing of a query, which is not a prepared statement, goes through several stages, which roughly are: parsing, optimization and execution. Prepared statements (PS), available since 4.1.0, are not parsed every time but once, when the PS is created. The optimization and execution phases are quite complex and also are out of the scope of this thesis; they will not be discussed but mentioned for completeness.

The parsing stage converts the textual representation of a query into internal format, which holds all information needed for the query execution. The MySQL query parser is reentrant and GNU Bison is used to build the parser (*sql/sql_yacc.cc*) from the grammar file (*sql/sql_yacc.yy*). GNU Bison is a parser generator similar to Yacc [Wiki]. Since Yacc is a LALR(1) scanner[LexYacc] thus Bison is also a LALR(1) (uses one token lookahead). A program is reentrant when it does not use global variables, hence not having critical sections, and can be executed in parallel. A non-reentrant parser will not fullfil the needs of the MySQL server, since the latter is a multithreaded program. The parser checks for the validity of a query and executes blocks of code, which handle specific branches in the query (the parser internally represents the parsing like a tree, by using stacks).

Every parser needs a lexer to be supplied with tokens. The lexer reads on demand from an input stream and recognizes patterns in it. When a pattern is matched, a token (structure) is returned to the parser. The grammar is based on tokens, which are the terminals of the grammar. A sequence of terminals and/or non-terminals is non-terminal.

"Typically, the LR / LALR parsing algorithms, like deterministic finite automata, are commonly represented by using a graph - albeit a more complex variant. For each token received from the scanner, the LR algorithm can take four different actions: Shift, Reduce, Accept and Goto. For each state, the LR algorithm checks the next token on the input queue against all tokens that are expected at that stage of the parse. If the token is expected, it is "shifted". This action represents moving the cursor past the current token. The token is removed from the input queue and pushed onto the parse stack."[LALR]. The same opinion is stated in [Dragon]. The latter mentions about third LR algorithm, canonical LR, which according to the authors of the book is the most powerful but still needs most resources. LALR parsing covers grammars that LR cannot cover and LALR is less computationally intensive than canonical LR. In addition, it is stated that "LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written. The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.".

Polymorphism is used extensively in the code that does the query processing. Every query is transformed in a tree with nodes ("items" is the term used by the documentation in the source code). All classes extend (subclass) *class Item* (*sql/item.h*). See Appendix C for class diagrams. For every MySQL function available in the queries there is respectively a class. The functions are separated into several groups depending on their return value:

- *class Item_str_func* is extended by all string functions
- Functions, which return numeric value descend from *class Item_real_func* or *class Item_int_func*.
- Functions, which return boolean value extend *class Item_bool_func*.

All non-functions in the query are also represented by instances of classes that subclass *class Item* (See Appendix C). The latter provides a large abstract interface implemented in the classes that extend it. More interesting methods from the interface are:

- `virtual enum Type type()`
- `virtual double val_real()`
- `virtual longlong val_int()`
- `virtual bool fix_fields(THD*, struct st_table_list*, Item**)`
- `virtual String *val_str(String*)`
- `virtual void print(String *str_arg)`
- `virtual bool get_date(TIME *ltime,uint fuzzydate);`
- `virtual bool get_time(TIME *ltime);`
- `virtual bool get_date_result(TIME *ltime, uint fuzzydate)`
- `virtual bool is_null()`
- `static CHARSET_INFO *default_charset()`

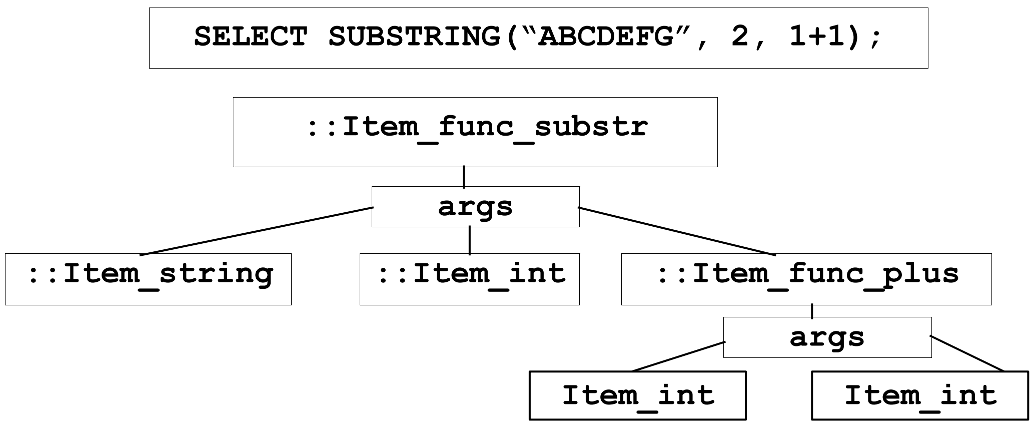


Figure 4.3 Parse tree of a simple SQL statement

*Item::fix_fields()* is used for fixation of fields. In some classes that extend it, more than just setting public variable *fixed* to 1 is done. If the

SQL statement shown in Fig. 4.3 included a FROM clause with a table and in the SUBSTRING() function a name of a column was used, then the fixation process would bind the instance of Item_field, which the parser will create, to a real field (of *class Field*) from the table (instance of *class TABLE*). If a string was used but it is not a field from the table an error will be emitted, because the fixation process is going to be unsuccessful. The second parameter to *fix_fields()* is a linked list with the tables used in the query. The field is looked amongst the fields of these. The third parameter is a Item** because if the Item has to be fixed the new pointer is returned there.

Example:

```
if (expr->fix_fields(thd, NULL, &expr))
   DBUG_RETURN(EVEX_PARSE_ERROR);
```

Like almost every other function in MySQL's source code, fix_fields() returns zero on success and non-zero on failure. The *Item_field* object contains column, table (optional) and database (optional) names. The field has to be bound to the real data source from which values will be fetched.

Many functions cannot work with an instance of a subclass of *class Item* if *fix_fields()* has not been invoked in advance. An overlook of *fix_fields()* call will lead to a debug assert being raised and the end of execution if the server is compiled in debug mode.

## 4.5 Storage engines

The server code that uses tables, like the optimizer or other parts responsible for data modification are not directly coupled with a low-level interface that performs the desired operations. This decoupling comes from the fact that there is an interface that exposes common operations like opening of a table, row read, row update, row deletion, sequential scans through the data file or sequential scan by using an index (if there is such), which is an ordered scan. The name of this handler interface is *HA*. More information the interface is available in [Pachev03]. This decoupling provides the opportunity of having different handlers, also known as storage engines, which are specialized in some way. At the moment of writing of this thesis the following storage engines (table types) exist:

- ISAM (deprecated in 3.23.00 and not available since 5.0.0). Non-transactional.
- MYISAM (default since 3.23.00). Non-transactional. Table level locking.
- MEMORY (previously known as HEAP). In-memory only. Hash based indexes. Since 4.1.0 also supports B-Tree indexes. Non-transactional. Data is lost when power is down. Table locking.
- InnoDB (available since 3.23 series). Transactional. Uses tablespaces. Row level locking.
- BDB (Berkeley DB). Available since 3.23 series. Transactional. Page level locking.
- NDB. Available since the 4.1 series. An interface to the cluster technology developed by MySQL AB.
- MERGE. Available since 3.23 series. Gives the opportunity to have a virtual table that merges tables equal in structure.
- Federated (In development). Gives the opportunity, to a user, to use other databases over protocols like JDBC, ODBC or proprietary, like standard MySQL tables.
- Access (In development): Read only access to MS Access MDB tablespaces. Useful for migration purposes.
- Paradox : Useful for migration purposes.
- Archive : Uses zlib, Lempel Ziv, compression.
- CSV : Stores the data in comma separated form. Used as an example storage handler.

## 4.6 System library

MySQL source is divided in several modules. One if this modules is the MySQL's system library which provides functions and classes, which are primitives, to be used in the whole server code. The code of the system library is spread over 2 directories in the source tree:

- strings/
- mysys/

The *strings/* directory contains several dozen files with usually one function per file. The functions are used for string operations. These functions should be used instead of the standard ones, which come with the standard C library for the platform for portability reasons. The functions are highly optimized for speed, even including portions of code implemented in assembler. It is the case that API of every operating system is different than the API of another. The POSIX specifications were created to standardize the APIs but still they are still different. Every multiplatform application uses some kind of system library, which abstracts from the underlying platform. An example product is APR (Apache Portable Runtime), http://apr.apache.org/, which is used by the Apache web server.

The *mysys/* directory contains more than 100 files with usually one function (or one functionality) per file. The following list is not by any means complete but shows what is available:

- dynamic sized arrays
- hashes
- double linked lists
- sorting (quick sort, radix sort)
- binary trees
- file operations
- command line options handling
- character sets handling
- memory allocation
- semaphores for platforms lacking implementation
- SHA1/MD5 hash functions and AES encryption

The dynamic arrays are very useful, because they provide functionality not available from the standard C arrays. The array may grow and shrink dynamically and the programmer does not need to handle the memory allocation, deallocation and reallocation, which is done behind the scenes by the library. The data is stored like normal C arrays in a sequential memory block, but additional info about how many elements are used is available. If an element is removed its position is reused by moving all elements with indexes greater than its own one position to the left. This leaves unused memory at the end of the block, which can be released with an API call. Also the array grows dynamically when it is completely full. The amount of memory preallocated is given during array creation.

The linked list in *mysys/* exports all functionality expected from such a structure, even including traversal with a callback function. However, another implementation exists, which is more like a container. It uses C++ templates and is in *sql/sql_list.h*. This implementation also provides traversal by means of iterators (Iterator is a Design Pattern specified in [GoF]).

## 4.7 Access control lists

To use and manipulate data in MySQL RDBMS different privileges and levels of privileges exists. Each one controls the access to different features and to all kind of objects that reside inside the system. The access control system is non-standard one. In short, it does not follow any of the SQL standards but is not complicated.

[MySQLRef] states "The primary function of the MySQL privilege system is to authenticate a user connecting from a given host, and to associate that user with privileges on a database such as SELECT, INSERT, UPDATE, and DELETE. Additional functionality includes the ability to have an anonymous user and to grant privileges for MySQL-specific functions such as LOAD DATA INFILE and administrative operations.".

The MySQL privilege system ensures that all users may perform only the operations that are granted to them. When a user connects to a MySQL server they are identified by the host from which the connection is performed and the user name specified. Privileges are granted from the system grants according to a specific identity and the intentions of the user.

MySQL considers both the hostname and the user name in the authentication process. The reason the hostname is taken into consideration is that users from different hosts may have the same user name. The database system can grant one set of privileges for connections from example.com, and a different set of privileges for connections from example.org, even when the user name is the same.

MySQL access control has two steps:

- The server checks whether the user is even allowed to connect. If not, the connection is aborted and statistical information about aborted connection attempts is updated.
- If the connection was not rejected on step 1, the server checks each statement issued to verify whether the user has sufficient privileges to perform it. For example, if one attempts to select rows from a table in a database or drop a table from the database, the access control system verifies that SELECT privilege had been granted to the user for the table or the DROP privilege for the database.

The command to show the privileges granted to a specific user is:

```
SHOW GRANTS FOR 'username'@'hostname';
```

The grants are separated in four levels (sql/sql_acl.h):

| | **GLOBAL** | **Database** | **Table** | **Column** |
|---|---|---|---|---|
| SELECT_ACL | + | + | + | + |
| INSERT_ACL | + | + | + | + |
| UPDATE_ACL | + | + | + | + |
| DELETE_ACL | + | + | + | + |
| CREATE_ACL | + | + | + | |
| DROP_ACL | + | + | + | |
| RELOAD_ACL | + | | | |
| FILE_ACL | + | | | |
| SHUTDOWN_ACL | + | | | |
| PROCESS_ACL | + | | | |
| GRANT_ACL | + | + | + | |
| REFERENCES_ACL | + | + | + | + |
| INDEX_ACL | + | + | + | |
| ALTER_ACL | + | + | + | |
| SHOW_DB | + | | | |
| SUPER_ACL | + | | | |
| CREATE_TMP_ACL | + | + | | |
| LOCK_TABLES_ACL | + | + | | |

| | GLOBAL | Database | Table | Column |
|---|---|---|---|---|
| REPL_SLAVE_ACL | + | | | |
| REPL_CLIENT_ACL | + | | | |
| EXECUTE_ACL | + | | | |
| CREATE_VIEW_ACL | + | + | + | |
| SHOW_VIEW_ACL | + | + | + | |

Table 4.1 MySQL privilege levels

## 4.8 Stored procedures

A stored procedure is a database object that is a collection of SQL statements that is referred under a name.

In SQL-99, which specifies a standard language for SP, they are called "SQL-Invoked Routines". This name is not broadly used because the term *stored procedures* has established itself as term for this entities.

[SQL99Compl] states : "A Schema may contain zero or more SQL-Invoked routines. An SQL-*invoked routine* (or SQL routine) is the generic name for either a procedure (SQL-invoked procedure) or a function (SQL-invoked function). SQL routines are dependent on some Schema (they're also called *Schema-level routines*) and are created, altered, and dropped using standard SQL statements.

The concepts of "procedure" and function" are the same in SQL as in other languages, so the ideas in this chapter will be old hat to any programmer. However, the syntax is all new; it will take time before all vendors fall in line – but it's certainly time that everybody know what routines are, according to the SQL standard."

All stored procedures and functions, which will be referred to as stored procedures, are stored in the *mysql* catalog. Every stored procedure can be executed at anytime and anywhere an SQL statement is allowed. A SP is executed by issuing the *CALL* SQL statement, according to the SQL standard [SQL99Compl] (BNF):

```
CALL <Routine name> <SQL argument list>
<SQL argument list> ::= ([ <SQL argument> [ {,<SQL argument>}...]])
<SQL argument> ::= scalar_expression_argument
            | :<host parameter name> [ [INDICATOR]:<host parameter
name>]
```

MySQL does not implement fully the BNF shown above as only the first part of the <*SQL argument*> non-terminal is implemented. However, the language used for SP is according to SQL3. This is an additional step forward toward SQL standard conformance. The implementation of SP is limited to what is relevant and possible in MySQL. SP in MySQL were introduced in version 5.0.0 (5.0.3-alpha is the current version at the time of writing of this thesis). Omitted from the implementation are the following SQL3 features [MySQLSP]:

- RESTRICT/CASCADE for ALTER/DROP
- METHODs (User Defined Types related) (MySQL does not support UDT).
- MODULEs (Related to Schema).

Stored procedures have both advantages and disadvantages. Most important ones are:

- SPs are closer to the RDBMS, thus saving communication overhead, which in turn costs time and resources. If the CPU power is unlimited, or there is unused one, this gives better performance when compared to a solution with an external program that uses the server's API (ODBC/JDBC or native) and fetches and sends data with calculation done at the client side. Following this strategy the clients are thinner and can be likened to dumb terminals.
- Being run on the server side imposes better control on what is accessed on a database level. Many big organizations impose strict rules regarding who accesses what. In such situations almost no one has access to the real data but the access is through defined stored procedures, which are known to be secure. For instance, a stored procedure may exist, which is used for withdrawal (as will be shown later) and direct access to the tables that contain the data is prohibited. Hence, a "curious" user will not be able to look directly at the customer data and their balance, nor the overall balance of the bank.
- Disadvantages include the fact that SPs are run on the server and the server must handle the CPU load they create. It is a known fact that the CPU power is not unlimited, thus the use of SPs imposes a bigger

load on the server machine.

- SPs may be used to avoid code duplication. By using a stored procedure for accessing data that spans over several tables the SELECT statement(s) is not copied over and over again but only the name of the procedure. If the underlying tables change only the SP has to be updated. This is similar to the idea of the VIEWs, even if the statements are UPDATE/DELETE (but if the VIEWs are updateable).

An example [MySQLSP] follows that shows how the operation "Withdrawal of money from a bank account" can be implemented as a stored procedure. The balance of all customers together is also updated.

```
CREATE PROCEDURE withdraw(
  p_amount DECIMAL(6,2),
  p_tellerid INTEGER,
  p_custid INTEGER)
MODIFIES SQL DATA
BEGIN ATOMIC
  UPDATE customers SET balance=balance - p_amount;
  UPDATE tellers SET cashonhand=cashonhand - p_amount
                 WHERE tellerid = p_tellerid;
  INSERT INTO transactions VALUES ( p_custid, p_tellerid, p_amount );
END;
```

Inside the MySQL server the stored procedures are saved in the table *proc* in the *mysql* catalog. All metadata needed for the compilation and execution is also stored there:

- database name (SPs are created per database)
- name (every stored procedure has an unique name inside the database where it is defined).
- type (PROCEDURE or FUNCTION).
- language (at the moment of writing only SQL)
- determinism (whether the SP is deterministic or not).
- security type ('INVOKER' or 'DEFINER'). Whose privileges are used when executing the routine. This applies also partly to the temporal triggers implemented as a prototype in this thesis. Because a user is not allowed to create a temporal trigger for another user therefore in the semantics of stored procedures this is security type DEFINER.
- parameters list
- body (as text)
- definer ('username'@'host')

- creation time
- last modification time
- SQL mode, a set of possible levels instructing the server which features of other RDBMSes to mimic.

Stored procedures are not kept on disk in compiled state but are compiled every time they are loaded from the database. In addition, the ALTER PROCEDURE statement does not allow the changing of the body of a routine. Therefore it is not necessary to recompile it and update it in the in-memory cache. The code that is produced during the compilation phase, when the CREATE PROCEDURE statement is parsed, is not native code but series of *instructions*. These instructions are later interpreted. In fact, every instruction is an object, which has an execute method which returns status code and an address, in case of jump. In case of jump, this address is of the next instruction to be executed. Example of compiled SP follows [MySQLSP]. In the left column the CREATE PROCEDURE statement is shown. In the right column the generated virtual machine instructions are presented:

| | |
|---|---|
| `CREATE PROCEDURE a(s CHAR(16))` | |
| `BEGIN` | |
| `    DECLARE x INT;` | |
| `    SET x = 3;` | `0: set(1,  3 )` |
| `    WHILE x > 0 DO` | `1: jump_if_not( 'x>0' , 5)` |
| `        SET x = x-1;` | `2: set(1, 'x-1' )` |
| `        INSERT INTO ...` | `3: statement( INSERT... )` |
| `    END WHILE;` | `4: jump(1)` |
| `END` | `5: <end>` |

When the variable *x* is declared, it is put into an array with all variables declared in the routine (spcont). Later, whenever possible the variable is referred by an ordinal number and not by name. The first elements in this array are the routine arguments. The counting starts from *0*. Therefore in the example above *x* is has ordinal number *1*, since number *0* has the procedure parameter *s*. For example, take a look at *instruction 0*, where *x* has been substituted with *1*.

The compiler makes two passes, which is quite normal. Algorithms for one pass compilers, while available, are not as straightforward as the one used. On the second pass, a process called *backpatching* is performed. It is needed since jumps, conditional and unconditional, are generated during the first compilation phase and have to be recalculated. In the example above, *instruction 1* has to be backpatched, because the address of the instruction immediately after the *END WHILE statement* is not known when the parser reads and generates an instruction for the *WHILE statement*. A LABEL statement is an example where an unconditional jump has to be backpatched during the second pass.

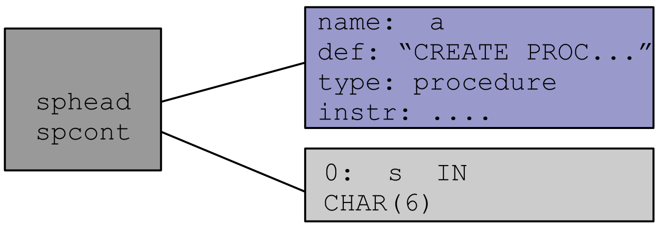


Figure 4.4 In memory SP structures

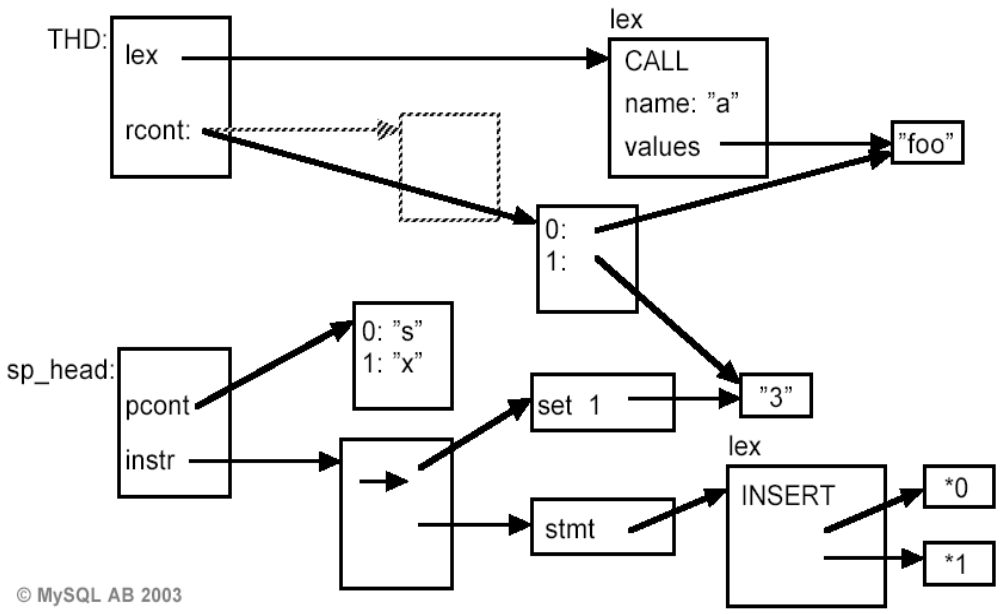


Figure 4.5 Memory structures during execution [MySQLSP]

When created, the stored procedures are not checked for semantic validity, but only syntax checking is performed. If a stored procedure refers to a non-existing table, its creation will not be aborted. However, the execution of the routine will fail if the table still does not exist at the time of execution. If a table is not referred to with a fully qualified name the interpreter assumes that it exists in the catalog where the routine is defined.

When a routine has to be executed a preparation phase is done. A few actions are performed (see also 4.5):

- The current catalog (database) in the thread descriptor (THD) is changed to the one of the routine. This leads to the fact that, if a table is referred to with short notation it is looked for in the database to which the stored procedure belongs.
- The routine is compiled if it is not cached already in memory. The compiled routines reside in their own memory area (pool) and are reachable from all threads inside the server. Nonetheless, the data segment is unique per thread, since two parallel executions of the same routine must not interfere with each other. Consequently, in the preparation phase the address for the variables array is set. The name of this array is *procedure context* or *rcont*. It is referred through the thread descriptor of *class THD*.
- Privileges are checked and set. If the security type is DEFINER the routine is executed with the rights of the user who defined the stored procedure, by means of all ACL levels listed in this document.

## 4.9 Table triggers

*"A Schema may contain zero or more Triggers. An SQL Trigger is a named chain reaction that you set off with an SQL-data change statement; it specifies a set of SQL statements that are to be executed (either once for each row or once for the whole triggering INSERT, DELETE, or UPDATE statement) either before or after rows are inserted into a Table, rows are deleted from a Table, or one or more Columns are update in rows of a Table. Triggers are dependent on some Schema – the <Trigger name> must be unique within the Schema to which the Trigger belongs – and are created, altered, and dropped using standard SQL statements."*[SQL99Compl].

Triggers in MySQL server completely rely on the recently introduced stored procedures to operate. When a trigger is defined it is checked for syntactic validity but nothing more. This is a consequence of the fact that the same parser, which parses stored procedures, is used for parsing the body of the trigger. In more detail, the same non-terminal <*sp_proc_stmt*> is used both by stored procedures and triggers. This non-terminal symbol can be used by any other code that needs the already existing functionality of stored procedures: parsing, compilation and execution. After it has been checked that a CREATE TRIGGER statement is valid, the whole SQL statement is written into a file, which contains all triggers for a specific table. At the time of writing of this thesis, this file has an extension .trg and is located in the data directory of the catalog. In MySQL catalogs are structured as subdirectories of the main data directory, on the file system. MyISAM tables exist in these subdirectories as .MYD(data) and .MYI(index) files. InnoDB tables and their indexes reside inside the InnoDB tablespaces. Nevertheless, in the catalog directory there is always a .frm file with the table definition. This file is used in the handler abstraction layer and the storage engines does nothing with it. Since the current format of the .frm file does not support storing trigger definitions, they are stored in an external file. When contacted, the author of the triggers implementation, Mr. Dmitri Lenev of MySQL AB, stated "in the future both will be merged together into one file". Current limitations are that up to six triggers can be defined per table. Respectively:

- For a pre-condition maximum one for each of INSERT, UPDATE and DELETE.
- For a post-condition maximum one for each of INSERT, UPDATE and DELETE.

Whenever a table is opened, the trigger definition file is opened and all existing triggers are loaded and compiled from there, by instantiating a parser and starting it. A consequence of this, mentioned in 4.7, is that the compiled state of the stored procedures reside in memory area common for all threads inside the server, the process of opening a table is synchronized with a mutex. In the current implementation, whenever a trigger on a table is created that table is closed. The reason is that next time this table is needed it will be opened and the newly defined trigger will be compiled and ready for use. As specified in SQL3, the triggers do not return a value. However if any of the statements that are executed as the body of a trigger fail, the trigger action will fail and no changes will be made.

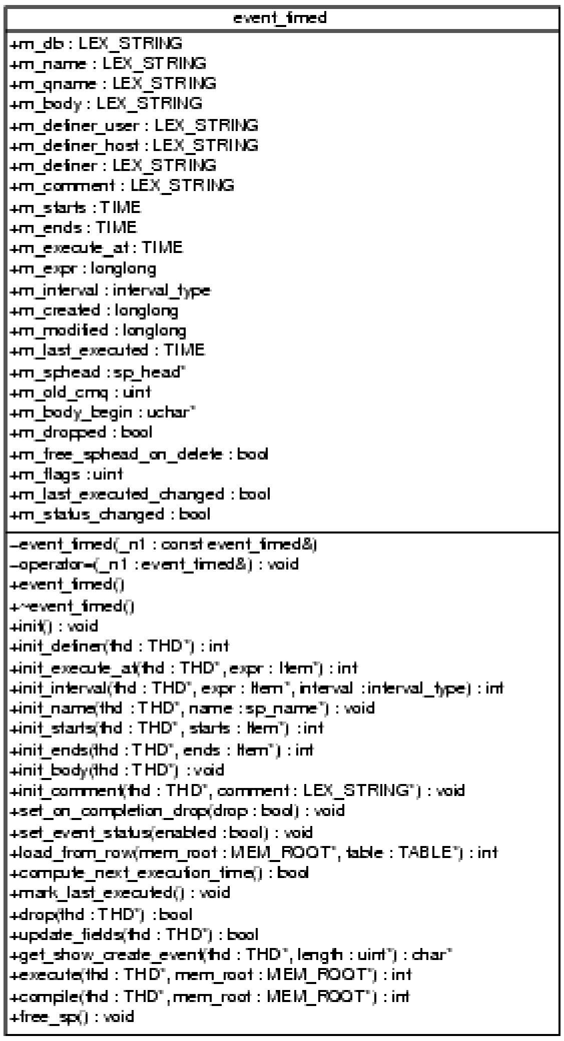
The way MySQL triggers behave is very similar to the behavior expected from temporal triggers.

# 5. Design of Low Level Architecture and Implementation

After thoroughly reviewing, in the previous chapter, different aspects from the MySQL implementation that are related to the implementation of temporal triggers, this chapter continues with implementation and low level architecture descriptions of the prototype.

## 5.1 Temporal triggers class

For the needs of temporal triggers a class named event_timed was created. It has the following definition:



```
event_timed
+m_db : LEX_STRING
+m_name : LEX_STRING
+m_qname : LEX_STRING
+m_body : LEX_STRING
+m_definer_user : LEX_STRING
+m_definer_host : LEX_STRING
+m_definer : LEX_STRING
+m_comment : LEX_STRING
+m_starts : TIME
+m_ends : TIME
+m_execute_at : TIME
+m_expr : longlong
+m_interval : interval_type
+m_created : longlong
+m_modified : longlong
+m_last_executed : TIME
+m_sphead : sp_head*
+m_old_cmq : uint
+m_body_begin : uchar*
+m_dropped : bool
+m_free_sphead_on_delete : bool
+m_flags : uint
+m_last_executed_changed : bool
+m_status_changed : bool
-event_timed(_n1 : const event_timed&)
-operator=(_n1 : event_timed&) : void
+event_timed()
+~event_timed()
+init() : void
+init_definer(thd : THD*) : int
+init_execute_at(thd : THD*, expr : Item*) : int
+init_interval(thd : THD*, expr : Item*, interval : interval_type) : int
+init_name(thd : THD*, name : sp_name*) : void
+init_starts(thd : THD*, starts : Item*) : int
+init_ends(thd : THD*, ends : Item*) : int
+init_body(thd : THD*) : void
+init_comment(thd : THD*, comment : LEX_STRING*) : void
+set_on_completion_drop(drop : bool) : void
+set_event_status(enabled : bool) : void
+load_from_row(mem_root : MEM_ROOT*, table : TABLE*) : int
+compute_next_execution_time() : bool
+mark_last_executed() : void
+drop(thd : THD*) : bool
+update_fields(thd : THD*) : bool
+get_show_create_event(thd : THD*, length : uint*) : char*
+execute(thd : THD*, mem_root : MEM_ROOT*) : int
+compile(thd : THD*, mem_root : MEM_ROOT*) : int
+free_sp() : void
```

Figure 5.1 Class diagram of *class event_timed*

The reason for having most of the properties public is that this speeds up operations. In any case, not having accessor methods has its drawbacks, like sometimes it is hard to find where a property is being changed. When an accessor method is used such a situation is very easy to track, by usage of a debugger or the debug log infrastructure of

MySQL.

The copy constructor is declared private, but not implemented, to prevent copying and passing an object by-value by mistake. This technique is mentioned by [EffectiveCPP]. If there was a standard or user implemented copy constructor, and by mistake an object is passed by value instead by reference (or pointer), the whole object has to be copied (cloned) by the copy constructor. When the object is big in size this can be slow and can lead to side effects like deallocation of memory which is still referenced. When passed by pointer only a pointer is copied, which is 4 or 8 bytes in size depending on the CPU architecture. In the case of *class event_timed,* the destructor destructs the instance of *class sp_head,* to which the temporal trigger holds a pointer. If the object is passed by reference, in the very moment when the code flow is leaving the visibility scope, the destructor of the copied object will be called and the m_sphead will be destructed, albeit that this object is still referenced from the original object.

All init_*() methods are used during the parsing process (see 5.2) to initialize the respective member variables of the object. In the first versions of the prototype, the real value a pointer to Item* created during the parsing process was held. However, this made *class event_timed* only suitable for parsing but not for object deserialization from disk. In addition, the *class Item* objects are destructed when the parser used for parsing the SQL statement is destructed. This leaves a *class event_timed* object with dangling pointers (pointers to deallocated memory). In addition, all init_*() methods call fix_fields() on the passed *class Item* pointer (see 4.4 for explanation why this step is important).

## 5.2 Parsing of SQL statements related to TTs

All reserved words in MySQL are listed in *sql/lex.h*, which is used later to generate a lexer based on a hash algorithm. The structure which keeps them is :

```
static SYMBOL symbols[]
```

The following reserved words were added (listed in alphabetical order):

- **AT** (*AT_SYM*)(related to transient events)
- **COMPLETION** (*COMPLETION_SYM*)
- **ENDS** (*ENDS_SYM*)
- **EVENT** (*EVENT_SYM*)
- **EVERY** (*EVERY_SYM*) (related to interval in reoccurring events)
- **PRESERVE** (*PRESERVE_SYM*)
- **SCHEDULE** (*SCHEDULE_SYM*)
- **STARTS** (*STARTS_SYM*)

In italics are shown the names of tokens, which are used in the parser grammar. These names follow a naming convention that suffixes the reserved keywords with the string *_SYM*. Nevertheless, there are still some exceptions of this rule, which are from the past when this rule had not existed. All these symbols are listed also in *sql/sql_yacc.yy*, which is the file containing the grammar. Every token has to be "registered" in the first section. The grammar file is separated in three sections. More information can be found in [LexYacc]. The tokens are registered this way:

```
%token  AT_SYM
%token  COMPLETION_SYM
%token  ENDS_SYM
%token  EVENT_SYM
%token  EVERY_SYM
%token  PRESERVE_SYM
%token  SCHEDULE_SYM
%token  STARTS_SYM
```

All these terminals were added to the non-terminal symbol *keyword* of the grammar, to allow users to use these as table, column, and database names. If these were not listed there their usage would be prohibited.

As well as the tokens, some non-terminal symbols, which are used later in the grammar, have to register in the same section. The idea is that a non-terminal may have associativity, which can be right or left. This is needed in cases where associativity matters, like in parsing arithmetical expressions.

In addition, non-terminals must be registered to declare their return value. The registration of all used non-terminal symbols follow:

```
%type <NONE> call sp_proc_stmts sp_proc_stmts1 sp_proc_stmt
%type <NONE> sp_proc_stmt_statement
%type <NONE> sp_proc_stmt_return  sp_proc_stmt_if
%type <NONE> sp_proc_stmt_case_no_expr  sp_proc_stmt_case_expr
%type <NONE> sp_proc_stmt_unlabeled_control sp_proc_stmt_leave
%type <NONE> sp_proc_stmt_iterate sp_proc_stmt_label
%type <NONE> sp_proc_stmt_goto sp_proc_stmt_open sp_proc_stmt_fetch
%type <NONE> sp_proc_stmt_close
```

The non-terminal *sp_proc_stmt* was changed to use the newly defined non-terminals, listed below, every single one of which is for only one "feature" of the stored procedures grammar. Because some non-terminals have no meaning in temporal trigger, like return of value and others, the new non-terminal symbols were introduced to be reused and avoid code duplication. New non-terminals:

- sp_proc_stmt_statement
- sp_proc_stmt_return
- sp_proc_stmt_if
- sp_proc_stmt_case_no_expr
- sp_proc_stmt_case_expr
- sp_labeled_control
- sp_proc_stmt_unlabeled_control
- sp_proc_stmt_leave
- sp_proc_stmt_iterate
- sp_proc_stmt_label
- sp_proc_stmt_goto
- sp_proc_stmt_open
- sp_proc_stmt_fetch
- sp_proc_stmt_close

The non-terminal used for the body of temporal trigger is named ev_sql_stmt_inner (all TT related non-terminals are prefixed with *ev_*):

```
ev_sql_stmt_inner:
          sp_proc_stmt_statement
        | sp_proc_stmt_return
        | sp_proc_stmt_if
        | sp_proc_stmt_case_no_expr
        | sp_proc_stmt_case_expr
        | sp_labeled_control {}
        | sp_proc_stmt_unlabeled_control
        | sp_proc_stmt_leave
        | sp_proc_stmt_iterate
        | sp_proc_stmt_label
        | sp_proc_stmt_goto
        | sp_proc_stmt_open
        | sp_proc_stmt_fetch
        | sp_proc_stmt_close
        ;
```

In *sql/sql_lex.h,* every server command processed in *sql/sql_parse.cc* is registered in *enum enum_sql_command*. New commands are added just before *SQLCOM_END*. The reason for doing so is that *enum*s in C/C++ have values starting from 0. If one additional last element is always present and its name is known, then the number of commands will be known. The same technique is used for counting number of fields in a table, for instance in *sql/sp.cc.* The new server commands are:

- SQLCOM_CREATE_EVENT
- SQLCOM_ALTER_EVENT
- SQLCOM_DROP_EVENT

In function *mysql_execute_command(),* located in *sql/sql_parse.cc*, dispatching according to *lex->sql_command* is done. An example of how SQLCOM_CREATE_EVENT is handled there, in a switch statement, follows:

```
switch (lex->sql_command) {
...
...
  case SQLCOM_CREATE_EVENT:
  {
    if (check_global_access(thd, EVENT_ACL))
      break;

    DBUG_ASSERT(lex->et);
    if (! lex->et->m_db.str)
    {
      my_message(ER_NO_DB_ERROR, ER(ER_NO_DB_ERROR), MYF(0));
      delete lex->et;
      lex->et= 0;
      goto error;
    }

    int result;
    uint create_options= lex->create_info.options;
    res= (result= evex_create_event(thd, lex->et, create_options));
    switch (result) {
      case EVEX_OK:
      send_ok(thd, 1);
      break;
    case EVEX_WRITE_ROW_FAILED:
      my_error(ER_EVENT_ALREADY_EXISTS, MYF(0), lex->et-
>m_name.str);
      break;
    case EVEX_NO_DB_ERROR:
      my_error(ER_BAD_DB_ERROR, MYF(0), lex->et->m_db.str);
      break;
    default:
      //includes EVEX_PARSE_ERROR
      my_error(ER_EVENT_STORE_FAILED, MYF(0), lex->et->m_name.str);
      break;
    }
    /*lex->unit.cleanup() is called outside,no need to call it
here*/
    delete lex->et;
    lex->et= 0;

    delete lex->sphead;
    lex->sphead= 0;

    break;
  }
...
...
}
```

The check whether the user can execute SQLCOM_CREATE_EVENT is performed by calling *check_global_access(thd, EVENT_ACL)*.

*evex_create_event()* (*sql/event.cc*) is the function which creates a new temporal trigger based on the information from the parser. The information is passed via the thread descriptor – thd, which contains a pointer to the parser, as well as the half constructed temporal trigger object *lex->et. my_error()* is used to send error messages to the client side in case of errors. In addition, all error messages can be translated in any language. Therefore, constants like ER_EVENT_STORE_FAILED, are used instead of literal strings containing error messages. The *evex_create_event()* function follows the rule for all functions in the serve code, as well as in libmysql, to return 0 as status code whenever a call is successful and return non-zero value when there is an error. *res* is a boolean variable on which mysql*_execute_command()* depends to find out whether the command execution was successful or not. The errors which are returned by all functions in the prototype are pre-processor macros and are defined in *sql/event.h*:

```
#define EVEX_OK                 0
#define EVEX_KEY_NOT_FOUND     -1
```

```
#define EVEX_OPEN_TABLE_FAILED -2
#define EVEX_WRITE_ROW_FAILED  -3
#define EVEX_DELETE_ROW_FAILED -4
#define EVEX_GET_FIELD_FAILED  -5
#define EVEX_PARSE_ERROR       -6
#define EVEX_INTERNAL_ERROR    -7
#define EVEX_NO_DB_ERROR       -8
#define EVEX_GENERAL_ERROR     -9
#define EVEX_BAD_PARAMS        -10
#define EVEX_NOT_RUNNING       -11
```

After *SQLCOM_CREATE_EVENT* is processed, the constructed temporal trigger object *lex->et* is destructed (memory is deallocated). Up in the call stack the memory pools are freed (see 4.3).

## 5.3 Metadata storage

The metadata is stored in a table named *event* in the *mysql* catalog. The table type (the table engine used) is MyISAM. The reasons for choosing this table type are:

- all other tables in the *mysql* catalog are of type MyISAM (privilege tables, time zones tables, stored procedures table).
- consequently MySQL server cannot boot up without having support for MyISAM, but still can operate without any other storage engine enabled.

The table definition is :

```
CREATE TABLE event (
  db varchar(64) character set latin1 collate latin1_bin NOT NULL
                      default '',
  name varchar(64) NOT NULL default '',
  body blob NOT NULL,
  definer varchar(77) character set latin1 collate latin1_bin NOT
NULL
                      default '',
  execute_at datetime default NULL,
  transient_expression int(11) default NULL,
  interval_type enum('YEAR','QUARTER','MONTH','DAY','HOUR',
                'MINUTE','WEEK','SECOND','MICROSECOND',
                'YEAR_MONTH','DAY_HOUR','DAY_MINUTE','DAY_SECOND',
                'HOUR_MINUTE','HOUR_SECOND','MINUTE_SECOND',
                'DAY_MICROSECOND','HOUR_MICROSECOND',
                'MINUTE_MICROSECOND','SECOND_MICROSECOND')
                 DEFAULT NULL,
  created timestamp NOT NULL default '0000-00-00 00:00:00',
  modified timestamp NOT NULL default '0000-00-00 00:00:00',
  last_executed datetime default NULL,
  starts datetime default NULL,
  ends datetime default NULL,
  status enum('ENABLED','DISABLED') NOT NULL default 'ENABLED',
  on_completion enum('DROP','PRESERVE') NOT NULL default 'DROP',
  comment varchar(64) character set latin1 collate latin1_bin NOT
NULL
                        default '',
  PRIMARY KEY  (db,name)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 COMMENT 'Events';
```

The temporal triggers prototype exports the following functions as an interface for creation, alteration and deletion:

```
int evex_create_event(THD *thd, event_timed *et, uint
create_options);

int evex_update_event(THD *thd, sp_name *name, event_timed *et);

int evex_drop_event(THD *thd, event_timed *et, bool drop_if_exists);
```

When data is parsed and *evex_create_event()* is called (which in turn calls db_create_event() to create the event on database level) and if EVEX(the executor thread) is running, the in-memory events cache is updated. The table is opened for writing this way:

```
  PRIMARY KEY  (db,name)"
  TABLE *table;
  TABLE_LIST tables;
  bzero(&tables, sizeof(tables));
  tables.db= (char*)"mysql";
  tables.real_name= tables.alias= (char*)"event";

  if (! (table= open_ltable(thd, &tables, TL_WRITE)))
  {
    ret= EVEX_OPEN_TABLE_FAILED;
    goto done;
  }
```

After that some of the needed values are written into the new row in the following way:

```
  restore_record(table, default_values); //Get default values for
fields
  ret= table->field[EVEX_FIELD_DEFINER]->
       store(definer, (uint)strlen(definer), system_charset_info);
  if (ret)
  {
    ret= EVEX_PARSE_ERROR;
    goto done;
  }

  ((Field_timestamp *)table->field[EVEX_FIELD_CREATED])->set_time();
  if ((ret= evex_fill_row(thd, table, et)))
    goto done;

  if (table->file->write_row(table->record[0]))
    ret= EVEX_WRITE_ROW_FAILED;

done:
  close_thread_tables(thd);
```

*evex_fill_row()* (*sql/event.cc*) is a function which is shared between SQLCOM_CREATE_EVENT and SQLCOM_ALTER_EVENT. Some of the fields written by these are the same and by using this function code duplication is prevented. *restore_record()* takes a TABLE pointer and restores (sets) the values of all columns in the current row to their default values. In the case of SQLCOM_CREATE_EVENT this is the desired behavior. After this call, all needed columns are updated with values according to the data from the parsing stage.

The way SQL_ALTER_EVENT works is quite similar (*created* column must not be updated but *modified*):

```
  TABLE *table;
  int ret;
  bool opened;
  ret= sp_db_find_entry(thd, 0/*notype*/, &et->m_name, &et->m_db,
                        TL_WRITE,
                        &table, &opened, (char*)"event",
                        &mysql_event_table_exists);
  if (ret == EVEX_OK)
  {
    store_record(table,record[1]);
    // Don't update create on row update.
    table->timestamp_field_type= TIMESTAMP_NO_AUTO_SET;
    ret= evex_fill_row(thd, table, et);
    if (ret)
      goto done;

    if (name)
    {
      table->field[EVEX_FIELD_DB]->
        store(name->m_db.str, name->m_db.length,
system_charset_info);
      table->field[EVEX_FIELD_NAME]->
        store(name->m_name.str,name-
>m_name.length,system_charset_info);
    }

    if ((table->file->update_row(table->record[1],table->record[0])))
      ret= EVEX_WRITE_ROW_FAILED;
  }
done:
  if (opened)
    close_thread_tables(thd);
  DBUG_RETURN(ret);
```

*sp_db_find_entry()* is a function in *sql/sp.cc,* which is used to find the row to be updated by using the primary key which is (*db, name*). Since stored procedures are similar, the code used by them was modified to be usable also by temporal triggers.

### 5.4 In-memory caching

To speed up execution of temporal triggers, instances of all triggers are kept in memory. In addition, because a temporal trigger may have status *DISABLED* only the triggers with status *ENABLED* are cached. In fact, there is no need to cache disabled events. On the other hand, if an event's status has been changed from *DISABLED* to *ENABLED* the temporal trigger will be loaded from disk and cached.

The cache consists of two dynamic arrays (*struct DYNAMIC_ARRAY*), see 4.6 for more info regarding MySQL dynamic arrays. The first dynamic array, *static DYNAMIC_ARRAY events_array*, contains all instances of *class event_timed* which have status ENABLED. Because this array is rather large, the smaller *static DYNAMIC_ARRAY evex_executing_queue* is used for scheduling. It holds pointers to the instances in the former array.

Whenever a new event is created with its status is set to *ENABLED* and the event executor is running (the main thread), the event will be cached in memory.

The data found during the parsing procedure will not be used during the caching. The fully qualified name of temporal trigger, which in fact is the primary key of table *event*, is used to load the metadata already written on disk. The position of the trigger in the *event* table will be searched with *sp_db_find_entry()* and when it is found the TABLE* pointer will be passed to the *event_timed::load_from_row()* method.

All allocations, except the two dynamic arrays, are done using a memory pool, owned by the temporal triggers module, and arte accessible throughout the whole module. In addition, the pool is not exposed to any other module directly.

The following code fragment shows the caching of triggers when they are created:

```
VOID(pthread_mutex_lock(&LOCK_evex_running));
if (!evex_is_running)
{
  VOID(pthread_mutex_unlock(&LOCK_evex_running));
  goto done;
  }
  VOID(pthread_mutex_unlock(&LOCK_evex_running));

  //cache only if the event is ENABLED
  if (et->m_status == MYSQL_EVENT_ENABLED)
  {
    spn= new sp_name(et->m_db, et->m_name);
    if ((ret= evex_load_and_compile_event(thd, spn, true)))
      goto done;
  }

done:
  if (spn)
    delete spn;
  DBUG_RETURN(ret);
```

*evex_load_and_compile_event()* is the function which, when passed a pointer to object of *class sp_name* (SP name), will find the event in the events table, load it into an object and then call *event_timed::compile()*. The latter, in turn, creates a new MySQL query parser and then starts it with a parameter that is a stripped down version of the CREATE EVENT statement used during the creation of the event. This stripped down version is used only for the compilaton of the temporal trigger's body. Eventually, there is a valid *class sp_head* object pointer, which resides in the memory pool of the stored procedures module. After successful compilation the parser is destroyed.

All manipulations on the two dynamic arrays are guarded by a mutex, namely *LOCK_event_arrays*. *evex_load_and_compile_event()* can be instructed not to acquire a lock on this mutex if the code which calls the function already holds it, because a second try to lock will lead to a deadlock or a crash. The crash occurs in debug build of MySQL because an assert will be triggered.

After the trigger's body has been compiled the next execution time is computed by calling *event_timed::compute_next_execution_time()*. After this is done, the execution queue will be sorted with qsort() (quick sort). The implementation is not the standard C library but a MySQL one, which can be found in *mysys/mf_qsort.c* . The double pointers to *class event_timed* objects are compared by using a comparator function, namely *static int event_timed_compare()*. In turn, this function calls *static inline int my_time_compare()*, which is part of the temporal triggers module, to compare the m_execute_at values of the two objects.

MySQL's standard library lacks a function to compare two values of struct TIME pointer. This was the reason this function is implemented, albeit MySQL has another one, named *TIME_to_ulonglong_datetime()*, which returns representation in longlong (64 bit) value which can be compared. *my_time_compare()* is used throughout the temporal triggers module whenever a comparison of two TIME values is needed, because it does not use multiplication as *TIME_to_ulonglong_datetime()* but only comparison.

## 5.5 Multithreading

The model used for execution of temporal triggers is master/slave. This pattern is widely known and sometimes referenced as “working crew” [POSIXThr].

The master is a thread that creates slave threads which perform work given them by the master thread. In the case of temporal triggers, the master thread looks into the events execution queue and if an event is found that is eligible for execution, a worker thread will be created with *posix_create()* and as parameter a pointer to an event_timed object will be passed. The pointer will be casted *void* *, because this is a requirement of *posix_create()*.

The preliminary implementation of this model (in the prototype) uses only one dynamic array; the one that holds all instances, and it is sorted every time an event has to be executed. In addition, only the first element from the queue was executed and then the queue was reordered. Still, one problem emerges when master/slave is implemented in this way: in the very moment after a new thread is created, the queue is sorted and the pointer to the temporal trigger which is passed to the slave thread becomes invalid immediately. In some cases, this may lead to inconsistent behavior, and in others to a server crash, because memory that has been freed will be accessed. The latter happens when the computation of the next execution time fails and thus the trigger is either disabled or dropped (depending on a clause at definition time). This is the case for transient events and events that have end_time set.

In the final implementation, as mentioned in section 5.4, two arrays are used and the slave holds a pointer to an object instance. The second array is used for scheduling and only this array is ordered. This array is used only for referencing the array with the triggers (the first dynamic array). Therefore the pointer becomes an invariant. A problem may occur when an event is being executed and in the meantime is dropped. In this case, the memory occupied by the event will be freed and the pointer will be dangling.

If a user deletes a trigger directly on the database level this will not lead to an error but the last execution time will not be recorded on disk.

The number of triggers executed in parallel may vary from platform to platform. On the platform used for development, which is SuSE Linux 9.2 with kernel Linux 2.6.4, the number is about 1020. However, this number also depends on the number of opened connections to the server, because for every new connection a new thread is spawned. It must be noted that this limitation is imposed by the POSIX threads implementation and not by the server code or the prototype. Successful long running tests were performed with about 100 temporal triggers scheduled for execution every second.

## 5.6 Statistical variables

In sql/mysqld.cc there is defined the variable *struct show_var_st status_vars[]*. In this variable all statistical variables are registered and shown with the command SHOW STATUS. For the prototype built for this thesis three variables were added to count the number of created, altered, and deleted events.

```
{"Com_alter_event",  (char*) offsetof(STATUS_VAR,
            com_stat[(uint)SQLCOM_ALTER_EVENT]),SHOW_LONG_STATUS},
{"Com_create_event", (char*) offsetof(STATUS_VAR,
            com_stat[(uint) SQLCOM_CREATE_EVENT]),
SHOW_LONG_STATUS},
{"Com_drop_event",   (char*) offsetof(STATUS_VAR,
            com_stat[(uint) SQLCOM_DROP_EVENT]), SHOW_LONG_STATUS}
```

The constants used to index array com_stat are from *enum enum_sql_command* (*sql/sql_lex.h*). These constants also count how many queries are executed per hour, because in MySQL the DBA can limit the number of queries per hour per user. Since these three commands change data (tables) they also have to be counted. The array with the counts is defined in *sql/sql_parse.cc* like this:

```
char  uc_update_queries[SQLCOM_END+1];
```

This array is updated in *mysql_execute_command()* (*sql/sql_parse.cc*) with :

```
statistic_increment(thd->status_var.com_stat[lex->sql_command],
                    &LOCK_status);
```

*lex* is an instance of *struct LEX* (which in turn is *typedef struct st_lex*), which is the parser used in MySQL. *struct st_lex* is defined in *sql/sql_lex.h*. This structure contains all information needed after parsing for the execution of the query. For example, one field is *sql_command*, which is used to identify what command has to be executed by the server. The type of the variable is *enum enum_sql_command.* In addition, *LOCK_status* is a mutex, which will be used to guard the update of the variable, passed as the first parameter. Every time a status variable is updated it is updated with this mutex locked.

## 5.7 Server variables

The behavior of the server is controlled by dozens of parameters, called variables. For instance, the server time zone is a server variable, which can be changed during run-time or even can be set when the server is started.

In the MySQL server there are two types of server variables, namely GLOBAL and SESSION. The GLOBAL variables affect the server as a whole, while SESSION variables are valid only for the current connection (thread). The time zone has dual nature. It is both a GLOBAL and a SESSION variable. Another example is *query_cache_type*, which controls the query caching mechanism in MySQL 4+ and has a dual nature.

Server wide (GLOBAL) variables descend from *class sys_var* (*sql/set_var.h*). Connection wide (SESSION) variables inherit from *class sys_var_thd* (*sql/set_var.h*), which in turn extends *class sys_var*. There are about 30 classes that extend these two. This number is high because every server variable has a type. Two of the available types are unsigned long and bool.

According to the HLA described in Chapter 3, a server variable must be implemented, which controls whether event execution is enabled or not. The name of the variable is *event_executor*. This variable is implemented as server wide (GLOBAL) and is of type bool, because this type fits our needs the best.

The definition of the variable is :

```
sys_var_bool_ptr sys_event_executor("event_executor",
                                     &event_executor_running_global_var);
```

The name of the object is *sys_event_executor*. The name of the global variable is *event_executor,* and is visible to all MySQL users. All changes affect the memory of the variable *event_executor_running_global_var*. The latter has type *my_bool* (which is *bool* in turn) and is declared in *sql/event.cc* and visible in *sql/set_var.cc* with an extern directive.

Whenever the value of *event_executor* needs to be checked for some value, *event_executor_global_var* holds the value (0 or 1) and is used. This variable forces the event executor thread to wait and not execute anything from the queue. The variable's value is checked two times per second. In short, there should not be too many checks per second, because this slows down the server, but on the other hand the server must not to wait too long when checking the variable.

## 5.8 Automatic testing

System testing has always been a part of the software development process, no matter what process is used. The difference between processes is usually "how & when". For the prototype of this thesis the "test often" approach was used. Instead of testing after the prototype is fully implemented, it was tested after every single feature added. Even more, the testing is automated by using the automated tests system of MySQL, which already exists and is used for running regression tests.

Regression testing is software testing that tries to reveal regression software defects, which occur whenever functionality that previously worked stops doing so. Often regression defects occur as an unintended consequence of changes in the software.

Common methods of regression testing are re-running previously run tests and checking whether previously fixed defects have reemerged. Experience shows that as software is developed, this kind of defect reemergence is unfortunately quite common. It is often the case that when some feature is redesigned, the same defects will be present in the redesign that were present in the previous design. Therefore the regression tests include functional tests as well tests for defects that have been fixed, which should not reemerge.

A snippet from the regression test for the prototype follows:

```
use test;
--disable_warnings
drop event if exists ppp222;
drop event if exists ppp224;
--enable_warnings

create event ppp222 on schedule every 15 minute
   starts now()
   ends date_add(now(), interval 5 hour)
   do call do_something();
drop event ppp222;

create event ppp224 on schedule every 15 minute
   starts now()
   ends date_add(now(), interval 5 hour)
   comment "some"
   do call do_something();
drop event ppp224;
```

The several statements above test the correct parsing of CREATE EVENT statements.

# 6. Conclusion

*Individual commitment to a group effort -- that is what makes a team work, a company work, a society work, a civilization work.*

*Vince Lombardi*i

As process automation progresses the need for task scheduling in MySQL has emerged.

The objective of this master thesis is the creation of a prototype that provides the functionality for scheduling of events and their execution, at specific moments in time, within the MySQL RDBMS.

The implementation of temporal triggers, or events/jobs, as they are referred in this thesis and other works, adds this feature to MySQL. The advantages provided by this functionality were already mentioned in Chapter 1, but briefly, temporal triggers are quite important in the areas of system administration and data warehousing.

This prototype's features have been discussed, during the writing of this thesis, with several software developers that use MySQL but have different backgrounds. Their opinions correlate in the point that this is a very nice addition to MySQL, and they would take advantage of it, if it were officially released as part of the product.

A partial problem during the research phase was that the only references found about temporal triggers, events or jobs, are in commercial product-specific manuals and books. No other works with scientific character, such as publications in scientific magazines, were found. The probable reason is that this topic is quite new (about 5 years old) in the area of relational database management and still has not been explored on a more scientific level. Similar to this are stored procedures, which have existed for years but have only been standardized in SQL3. Moreover, the author of this thesis did not find any implementation of this feature in an *Open Source* RBDMS. Ergo, there was no way to compare the prototype built for this thesis with existing open implementations.

It must be taken into consideration that the author lacks low-level knowledge about the different parts and the inner workings of the MySQL

RDBMS, leading to the system being studied on the fly. This might be considered a shortcoming but large systems such as MySQL cannot be studied in a matter of days or weeks, which are the time constraints of a master thesis.

Something that deserves to be noted is that while the MySQL RDBMS is implemented mostly in C++, it does not use C++ exception handling. The reason for not doing so is because MySQL strives to be compilable on a vast range of platforms; as explained in the internal documentation. It was found that some platforms does not have very good C++ implementation and exception handling was not working as expected. Consequently, the system does not use exceptions but gotos. The usage of the latter may look frightening considering the famous work of E. Dijkstra, "GO TO statement considered harmful". However, when used properly GOTO emulates the behavior of exceptions and has an advantage of being faster, something which must be considered in such a system like a RDBMS, which should be as fast as possible.

In addition, because of the time constraints a few requirements, which appear in Chapter 3, were not implemented. This issue was discussed with the technical advisor from MySQL AB and Mr. Brian Aker, also from the same company, and the decision is that their omission is not critical. These are:

- logging the result/output to a text file
- conditional compilation of the module
- FLUSH EVENTS command to clear the cache.
- serialized event execution
- the ability for a temporal trigger to create/alter/drop another temporal trigger (because the values stored in thd->lex are not reentrant). Temporal triggers cannot contain other temporal triggers.

In conclusion, MySQL AB, the company behind MySQL RBDMS, intends to include the presented prototype in a future MySQL version, if the code review process is successful. MySQL AB uses code reviews as a methodology for improving source code quality and minimizing defects. If the prototype is approved for inclusion in the official source code versioning system, its functionality will be expanded and many of the shortcomings, which emerged because of the time constraints, will be fixed. At first, the unimplemented features listed above will be developed.

# Appendices

## Appendix A. Glossary

- **API** (Application Programming Interface): Specifications for accessing common sets of functionality. Such interfaces are being created to abstract the access to a specific system and present the latter as a black-box which is controlled only by the API. API is close to the Facade design pattern.
- **BNF** (Backus Naur Form): "The Backus-Naur form (also known as Backus normal form) is a metasyntax used to express context-free grammars: that is, a formal way to describe formal languages. BNF is widely used as a notation for the grammars of computer programming languages, command sets and communication protocols"[Wiki].
- **command interpreter**: interactive program that is an interface between an user and an OS. Also known as a **shell**.
- **data mart** : "A subset of a data warehouse that contains data that is tailored and optimized for the specific reporting needs of a department or team. A data mart can be a subset of a warehouse for an entire organization, such as data that is contained in online analytical processing (OLAP) tools "[DB2G].
- **data warehouse** : "A subject-oriented nonvolatile collection of data that is used to support strategic decision making. The warehouse is the central point of data integration for business intelligence (BI). It is the source of data for data marts within an enterprise and delivers a common view of enterprise data"[DB2G].
- **daylight saving time** (DST): "The local time a region is designated for a portion of the year, usually an hour forward from its standard official time. Also known as Summer Time. All countries in Europe, except Iceland, observe DST and switch at the same universal time (1:00 UTC) in all five zones, going from 10pm/0/1/2/3am LST to 11pm/1/2/3/4am LDT simultaneously on the last Sunday in March, and back from 11pm/1/2/3/4am LDT to 10pm/0/1/2/3am LST on the last Sunday in October (formerly September) (for the European Union, except the overseas territories, per EU directive 2000/84/EC for some

of Greenland: the Saturday before).”[1]

- **DBA** (data base Administrator): A person who is responsible for the design, development, operation, security, maintenance, and use of a database.
- **DDL** (Data Definition Language): A language for describing data and its relationships in a database.
- **design pattern**: A specific way of solving of a specific problem in the application development. There are several catalogs of design patterns specific to different areas of software development. Design patterns make creation of software more of an engineering practice. The first book on the topic is [GoF].
- **DLL** (Dynamic link library): A chunk of machine code that is not standalone executable but is loaded at a program's runtime to load additional functionality. The term is popular on Windows OS.
- **DML** (Data Manipulation Language): A subset of SQL statements that are used to manipulate data. Most applications primarily use DML SQL statements.
- **EVEX** (EVent EXecutor): A subsystem of the prototype, created for the master thesis, which is responsible for execution of defined events.
- **FIFO** (First In First Out): Discipline showing data flow. Used in implementation of the *queue* data structure. The first data entered into the queue will be the first which will be fetched. The opposite is LIFO.
- **GUI** (Graphical User Interface): User interface that is based on graphics. On the contrary is TUI (Text User Interface), which is less common nowadays.
- **LAN** (Local Area Network): “A local area network (LAN) is a computer network covering a local area, like a home, office or small group of buildings such as a college. The topology of a network dictates its physical structure.”[Wiki]
- **LIFO** (Last In First Out): Discipline like FIFO, but used for data structure stack. The last data stored in the data structure is the first to be fetched.

[1] http://en.wikipedia.org/wiki/Daylight_Savings

- **LALR (n)**: Lookahead Left to Right parsing. *n* is the number of tokens, which are read in ahead whenever needed. LALR is and extension of LR.
- **linting** : Comes from the name of a popular tool that checks the syntax validity of C/C++ programs without the phases of compilation and object linking into binary code.
- **makefile** : A file with instructions for how to compile. Consists of rules that specify the order of compilation as well as possible execution of external entities. A makefile is a configuration for compiling a program or a module. The makefile is interpreted by the *make* program for which there are different flavors. A parallel are project files in the Microsoft Visual Studio IDE or in similar programs as well as Apache Ant which is used for the compilation of programs written in Java.
- **MAN** (Metropolitan Area Network): "Metropolitan area networks or MANs are large computer networks usually spanning a campus or a city. They typically use optical fiber connections to link their sites."[Wiki]
- **master** : One of the roles in a MySQL one-way replication mechanism. The second role is the slave. A master records all SQL statements that change data in its logs. These logs are used by the replication mechanism whenever a replica (slave) connects to the master and asks for the delta of statements that has been logged since the last time statements were fetched by the slave.
- **multi-master** : Replication topology where a RDBMS is both a master and a slave. For instance if there are 2 machines A and B and data&structure changing statements are executed on both A and B, then B replicates from A and respectively A from B.
- **non-terminal**: In Backus Naur Form (BNF) a symbol that is expressible in terms of other terminals or non-terminals. A recursive term.
- **ODBMS** (Object Database Management System): usually a client-server implemented system that stores every single entity as an object and provides access to stored data by means of OQL.

- **OLAP** (On-Line Analytical Processing): A term used in Data Warehousing. "Usually a multidimensional, multi-user, client server computing environment for users who need to analyze consolidated enterprise data in real time. OLAP systems feature zooming, data pivoting, complex calculations, trend analysis, and modeling"[DB2G].
- **OLTP** (On-Line Transactional Processing): The processing of transactions by computers in real time.
- **OS** (Operating System) A layer between the hardware and the applications (software) that is essential and provides an abstraction. If an OS runs on different kinds of hardware it is likely that the software will continue to work when moved from one hardware platform to another either in binary form or by recompiling the application. OS provides an API that is used by the programs to abstract working with the hardware.
- **OQL** (Object Query Language): the language used for querying ODBMS.
- **RDBMS** (Relational Database Management System): usually a client-server technology implemented system that provides functionality for accessing data stored in a relational way as specified by Codd. The usual way to work with a RDBMS is to use SQL. The relational model is different than the network and hierarchical models.
- **recurring event**: Event that is being executed more than once. Its schedule allows more than one execution. An example is an event that is executed every 60 minutes.
- **regression testing** : Software testing which tries to reveal regression software defects. Regression defects occur whenever functionality that previously worked stops doing so.
- **replica** : See *slave*.
- **service**: a highly specific application that resides on top of an OS and provides functionality that extends that of the OS. Sometimes called a daemon.
- **shared memory**: Memory segments mapped directly into the address space of a process. The OS kernel does the mapping during read/write

operations.

- **shell** : See *command interpreter*.
- **shell script** : Ordered set of instructions interpreted and executed by a  command interpreter (shell).
- **slave** : See *master*.
- **SO** (Shared Object): Equivalent of a Windows DLL in *nix operating systems.
- **SP** (Stored Procedure): A program that is saved in and executed by a RDBMS. Defined in the SQL-99 standard. An advantage is that the program is closer to the RDBMS and reduces the client/server communication overhead, although the execution increases the CPU load of the database server and hardware.
- **SSL** (Secure Sockets Layer): "SSL and Transport Layer Security (TLS), its successor, are cryptographic protocols which provide secure communications on the Internet."[Wiki]
- **SQL**  (Structured Query Language): A standardized language for defining and manipulating data in a relational database. It does not specify how to extract the information needed but only what information is needed. This is different than the way network and hierarchical database systems are programmed. SQL gives a level of abstraction over the internal structures of the data stored. Extensions are OQL and SQL with object extensions.
- **temporal trigger** (TT) : A non table-based trigger executed according to a schedule.
- **terminal** (symbol) : A terminal symbol, in BNF jargon, is a symbol that represents a constant value.
- **transient event** : An event that occurs only once. An example is an event that is scheduled for execution at 2004-12-26 15:00:00.
- **trigger** : An object in a database that is invoked indirectly by the database manager when a particular SQL statement is run. Table trigger types are: ON INSERT, ON UPDATE, ON DELETE and the execution can be scheduled either before or after the event that triggers the trigger's execution is performed. The granularity can be

*per row* or *once for the entire statement*.

- **UDF** (User Defined Function): A function that is written (usually in C/C++) and can be loaded at runtime into MySQL as additional functionality. UDFs are grouped in binary files that are similar to DLL/SO dynamic loadable libraries.
- **UTC** (Universal Coordinated Time): A time zone that never has daylight saving and is the basis for calculation of other time zones. UTC is the successor of Greenwich Mean Time, abbreviated as GMT, and may still colloquially be called GMT on occasion.
- **WAN** (Wide Arean Network): "A wide area network or WAN is a computer network covering a wide geographical area, involving vast array of computers. This is different from personal area networks (PANs), metropolitan area networks (MANs) or local area networks (LANs) that are usually limited to a room or a building. The best example of a WAN is the Internet."[Wiki]

## Appendix B. References

[AlgHeap] J. W. J. Williams. *Algorithm 232 Heapsor,* "Communications of the ACM", 7(6), p347-348, 1964.

[ASIQPG] "Adaptive Server IQ Administration and Performance Guide", Chapter 17 "Automating Tasks Using Schedules and Events".

[ASIQRM] "Adaptive Server® IQ Reference Manual", Adaptive Server® IQ 12.5, DOCUMENT ID: 38151-01-1250-02

[Beck04] "Extreme Programming Explained", K. Beck, D. Andres, "Addison-Wesley Publishing Company", 1999, ISBN 0-201-616416.

[DADS] "Dictionary of Algorithms and Data Structures", Jan 21st,
http://www.nist.gov/dads/

[DBAC] "Database Access", DB2 Developer Domain, Sept 15th
http://www7b.software.ibm.com/dmdd/

[DB2G] "IBM DB2 Universal Database Glossary", Version 8

[Dragon] "Compilers: Principles, Techniques and Tools", Alfred Aho, Ravi Sethi, Jeffrey Ullman, "Addison-Wesley Publishing Company", 1986, ISBN 0-201-10088-6

[EffectiveCPP] "Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Second Edition", Scott Meyers, Addison-Wesley, 1997

[EBNF] ISO/IEC 14977:1996(E) , August 24th,
http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf

[EBNF2] A Summary of the ISO EBNF Notation, Markus Kuhn, August 24th, http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html

[Fierros03] “An Introduction to DB2 UDB Scripting on Windows”, Chris Fierros, 2003, Sept. 15th,
http://www-106.ibm.com/developerworks/db2/library/techarticle/0307fierros/0307fierros.html

[GoF] “Design Patterns: Elements of reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Addison-Wesley Publishing Company”, 1996 , ISBN 0-20-16336-12

[GOTO] “Go To statement considered harmful”, Edsger W. Dijkstra, “Communications of the ACM”, Vol. 11, No. 3, March 1968, pp. 147-148.

[LexYacc] “Lex & Yacc”, Second Edition, John Levine, Tony Mason, Doug Brown, “O'Reilly & Associates Inc.”, ISBN 1-56592-000-7

[JSUG] “Jobs Scheduler User's Guide, Adaptive Server Enterprise 12.5.2”, Document ID: DC20001-01-1252-01 (Official Sybase Inc. documentation).

[LALR] “GOLD Parser : Left-to-Right Parsing”, Jan. 23rd ,
http://www.devincook.com/goldparser/concepts/lalr.htm

[MSDN] “MSDN documentation”, Sept. 12th :
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_sp_00_519s.asp

[MySQLRef] “MySQL Reference Manual”, Aug 23rd,
http://dev.mysql.com/doc/

[MySQLSP] “Stored Procedures in MySQL 5.0”, Per-Erik Martin, Presentation at “MySQL User Conference 2003”.
ftp://netmirror.org/mysql.com/Downloads/Presentations/MySQL-User-Conference-2003/MySQL-Stored-Procedures.pdf

[ODAG] “Oracle® Database Administrator's Guide 10g Release 1 (10.1)” Part No. B10739-01 December 2003. Chapters 26, 27 & 28.

[ODNF] “Oracle Database 10g New Features” , Robert Freeman, “Osborne/McGraw-Hill”, 2004, ISBN: 0072229470

[Pachev03] “MySQL Enterprise Solution”, Chapter 18, Alexander Pachev, “Wiley, John & Sons, Incorporated”, 2003, ISBN 0471269220

[POSIXThr] “Programming with POSIX threads”, David R. Butenhof, “Addison-Wesley Longman Inc.”, 1997, ISBN 0-201-63392-2

[SQL99Compl] “SQL-99 Complete, Really”, Peter Gulutzan, Trudy Peltzer, “R&D Books”, ISBN 0879305681.

[WikiSyb] “Sybase”: Wikipedia, the free encyclopedia, Jan 15th : http://en.wikipedia.org/wiki/Sybase

[WikiCS] “Client/Server”: Wikipedia, the free encyclopedia, Jan 17th : http://en.wikipedia.org/wiki/Client_server

## Appendix C. MySQL server class diagrams

*I hereby declare that I have written this work independently and used no resources other than the indicated aids.*

*Stuttgart*

***February 12, 2005***

*..................................*

*Andrey Jordanov Hristov*